Embedded System Design:

Topics, Techniques and Trends

Edited by
Achim Rettberg
Mauro C. Zanella
Rainer Dömer
Andreas Gerstlauer
Franz J.Rammig

Springer

ifip

EMBEDDED SYSTEM DESIGN: TOPICS, TECHNIQUES AND TRENDS

IFIP – The International Federation for Information Processing

IFIP was founded in 1960 under the auspices of UNESCO, following the First World Computer Congress held in Paris the previous year. An umbrella organization for societies working in information processing, IFIP's aim is two-fold: to support information processing within its member countries and to encourage technology transfer to developing nations. As its mission statement clearly states,

> *IFIP's mission is to be the leading, truly international, apolitical organization which encourages and assists in the development, exploitation and application of information technology for the benefit of all people.*

IFIP is a non-profitmaking organization, run almost solely by 2500 volunteers. It operates through a number of technical committees, which organize events and publications. IFIP's events range from an international congress to local seminars, but the most important are:

- The IFIP World Computer Congress, held every second year;
- Open conferences;
- Working conferences.

The flagship event is the IFIP World Computer Congress, at which both invited and contributed papers are presented. Contributed papers are rigorously refereed and the rejection rate is high.

As with the Congress, participation in the open conferences is open to all and papers may be invited or submitted. Again, submitted papers are stringently refereed.

The working conferences are structured differently. They are usually run by a working group and attendance is small and by invitation only. Their purpose is to create an atmosphere conducive to innovation and development. Refereeing is less rigorous and papers are subjected to extensive group discussion.

Publications arising from IFIP events vary. The papers presented at the IFIP World Computer Congress and at open conferences are published as conference proceedings, while the results of the working conferences are often published as collections of selected and edited papers.

Any national society whose primary activity is in information may apply to become a full member of IFIP, although full membership is restricted to one society per country. Full members are entitled to vote at the annual General Assembly, National societies preferring a less committed involvement may apply for associate or corresponding membership. Associate members enjoy the same benefits as full members, but without voting rights. Corresponding members are not represented in IFIP bodies. Affiliated membership is open to non-national societies, and individual and honorary membership schemes are also offered.

EMBEDDED SYSTEM DESIGN: TOPICS, TECHNIQUES AND TRENDS

IFIP TC10 Working Conference: International Embedded Systems Symposium (IESS), May 30 – June 1, 2007, Irvine (CA), USA

Edited by

Achim Rettberg
Paderborn University/ C-LAB
Germany

Mauro C. Zanella
ZF Lemförder GmbH
Germany

Rainer Dömer
University of California, Irvine
USA

Andreas Gerstlauer
University of California, Irvine
USA

Franz J. Rammig
Paderborn University
Germany

Springer

Library of Congress Control Number: 2007925211

Embedded System Design: Topics, Techniques and Trends

Edited by A. Rettberg, M. Zanella, R. Dömer, A. Gerstlauer, and F. Rammig

p. cm. (IFIP International Federation for Information Processing, a Springer Series in Computer Science)

ISSN: 1571-5736 / 1861-2288 (Internet)
ISBN: 13: 978-0-387-72257-3
eISBN: 13: 978-0-387-72258-0
Printed on acid-free paper

Copyright © 2007 by International Federation for Information Processing.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

9 8 7 6 5 4 3 2 1
springer.com

Contents

2 Automotive Applications

3 Hardware Synthesis

Preface

This book presents the technical program of the International Embedded Systems Symposium (IESS) 2007. Timely topics, techniques and trends in embedded system design are covered by the chapters in this book, including design methodology, specification and modeling, embedded software and hardware synthesis, networks-on-chip, distributed and networked systems, and system verification and validation. Particular emphasis is paid to automotive and medical applications. A set of actual case studies and special aspects in embedded system design are included as well.

Over recent years, *embedded systems* have gained an enormous amount of processing power and functionality. Many of the formerly external components can now be integrated into a single System-on-Chip. This tendency has resulted in a dramatic reduction in the size and cost of embedded systems. As a unique technology, the design of embedded systems is an essential element of many innovations.

Embedded systems meet their performance goals, including real-time constraints, through a combination of special-purpose hardware and software components tailored to the system requirements. Both the development of new features and the reuse of existing intellectual property components are essential to keeping up with ever demanding customer requirements. Furthermore, design complexities are steadily growing with an increasing number of components that have to cooperate properly. Embedded system designers have to cope with multiple goals and constraints simultaneously, including timing, power, reliability, dependability, maintenance, packaging and, last but not least, price.

The significance of these constraints varies depending on the application area a system is targeted for. Typical embedded applications include multi-media, automotive, medical, and communication devices.

The *International Embedded Systems Symposium (IESS)* is a unique forum to present novel ideas, exchange timely research results, and discuss the state of the art and future trends in the field of embedded systems. Contributors and participants from both industry and academia take active part in this symposium. The IESS conference is organized by the Computer Systems Technology committee (TC10) of the International Federation for Information Processing (IFIP).

IESS is a true inter-disciplinary conference on the design of embedded systems. Computer Science and Electrical Engineering are the predominant academic disciplines concerned with the topics covered in IESS, but many applications also involve civil, mechanical, aerospace, and automotive engineering, as well as various medical disciplines.

In 2005, IESS was held for the first time in Manaus, Brazil. In this initial installment, IESS 2005 was very successful with 30 accepted papers ranging from specification to embedded systems application. *IESS 2007* is the second installment of this conference, establishing a regular bi-annual schedule.

IESS 2007 is held in Irvine, California, at the Beckman Conference Center of the National Academies of Sciences and Engineering. The conference center is located on the campus of the University of California at Irvine. Co-located with one of the leading research institutions on embedded systems, the Center for Embedded Computer Systems at UC Irvine, this IESS conference is a unique venue to establish and foster research relations and collaboration between academic researchers and industry representatives worldwide.

The articles presented in this book are the result of a rigorous double-blind review process implemented by the technical program committee. Out of a total of 64 valid submissions, 35 papers have been accepted for publication, yielding an overall acceptance rate of 54.7%.

Confident about a strong technical program, we look forward to a fruit- and successful IESS 2007 conference,

Achim Rettberg, Mauro C. Zanella, Franz J. Rammig, Rainer Dömer, and Andreas Gerstlauer

Irvine, California, March 2007

Acknowledgements

First and foremost, we thank our sponsors ZF Lemförder GmbH, the Office of Research at the University of California at Irvine (UCI), and the Center for Embedded Computer Systems (CECS) at UCI for their generous financial support of this conference. Without these contributions, IESS 2007 would not have been possible in its current form.

We would also like to thank IFIP as organizational body for the promotion and support of the IESS conference.

Last but not least, we thank the authors for their interesting research contributions and the members of the technical program committee for their valuable time and effort in reviewing the articles.

IFIP TC10 Working Conference:
International Embedded Systems Symposium (IESS)
May 30 – June 1, 2007
Irvine, California

General Chairs
Achim Rettberg
Mauro C. Zanella
General Co-Chair
Franz J. Rammig
Technical Program Chair
Rainer Dömer
Local Arrangements Chair
Andreas Gerstlauer

Technical Program Committee
Richard Anthony – The University of Greenwich, England
Samar Abdi – University of California at Irvine, USA
Jürgen Becker – University of Karlsruhe, Germany
Brandon Blodget – Xilinx Research Labs, USA
Christophe Bobda – University of Kaiserslautern, Germany
Rainer Doemer – University of California at Irvine, USA
Cecilia Ekelin – Volvo Technology Corporation, Sweden
Rolf Ernst – Technical University of Braunschweig, Germany
Masahiro Fujita – University of Tokyo, Japan
Andreas Gerstlauer – University of California at Irvine, USA
Frank Hansen – Altera, Germany
Joerg Henkel – University of Karlsruhe, Germany
Thomas Heurung – Mentor Graphics, Germany
Uwe Honekamp – Vector Informatik, Germany
Marcel Jackowski – USP, Brazil
Kane Kim – University of California at Irvine, USA
Bernd Kleinjohann – C-LAB, Germany

Thorsten Koelzow – Audi Electronics Venture, Germany
Hermann Kopetz – Technical University of Vienna, Austria
Horst Krimmel – ZF Friedrichshafen, Germany
Jean-Claude Laprie – LAAS, France
Thomas Lehmann – Philips, Germany
Roger May – Altera, England
Mike Olivarez – Freescale Semiconductor, USA
Frank Oppenheimer – OFFIS, Germany
Carlos Pereira – UFRGS, Brazil
Franz Rammig – University of Paderborn, Germany
Achim Rettberg – C-LAB, Germany
Carsten Rust – Sagem Orga, Germany
Stefan Schimpf – Robert Bosch Ltda., Brazil
Juergen Schirmer – Robert Bosch GmbH, Stuttgart, Germany
Aviral Shrivastava – Arizona State University, USA
Joachim Stroop – dSPACE, Germany
Hiroyuki Tomiyama – Nagoya University, Japan
Ansgar Traechtler – University of Paderborn, Germany
Flavio R. Wagner – UFRGS, Brazil
Mauro Zanella – ZF Lemförder, Germany
Jianwen Zhu – University of Toronto, Canada

Organizing Committee

Achim Rettberg, Mauro C. Zanella, Rainer Dömer, Andreas Gerstlauer

Co-Organizing Institution

IFIP TC 10, WG 10.5 and WG 10.2

Sponsors

ZF Lemförder GmbH
Office of Research, University of California, Irvine
Center for Embedded Computer Systems, UC Irvine

REQUIREMENTS AND CONCEPTS FOR TRANSACTION LEVEL ASSERTION REFINEMENT

Wolfgang Ecker
Infineon Technologies AG
IFAG COM BTS MT SD
81726 Munich, Germany
Wolfgang.Ecker@infineon.com

Volkan Esen, Thomas Steininger, Michael Velten
Infineon Technologies AG
TU Darmstadt - MES
Firstname.Lastname@infineon.com

Abstract: Both hardware design and verification methodologies show a trend towards abstraction levels higher than RTL, referred to as transaction level (TL). Transaction level models (TLMs) are mostly used for early prototyping and as reference models for the verification of the derived RTL designs. Assertion based verification (ABV), a well known methodology for RTL models, has started to be applied on TL as well. The reuse of existing TL assertions for RTL and/or mixed level designs will especially aid in ensuring the functional equivalence of a reference TLM and the corresponding RTL design. Since the underlying synchronization paradigms of TL and RTL differ - transaction events for TL, clock signals for RTL - a direct reuse of these assertions is not possible. Currently there is no established methodology for refining the abstraction of assertions from TL towards RTL. In this paper we discuss the problems arising when refining TL assertions towards RTL, and derive basic requirements for a systematic refinement methodology. Building on top of an existing assertion language, we discuss some additional features for the refinement process, as well as some examples to clarify the steps involved.

Keywords: ABV, Mixed-level Assertions, TL Assertions, Assertion Refinement

1. INTRODUCTION

Early prototyping with the usage of higher abstraction levels than RTL has become increasingly popular during recent years [4] and is used more and more in industrial workflows. The most common modeling paradigm in this regard

Ecker, W., Esen, V., Steininger, T., Velten, M, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 1–14.

is transaction level modeling. Transaction Level models (TLMs) are used for architecture exploration, as early platforms for software development, and later on as golden reference models for verification of the corresponding RTL designs. Besides, using so called transactors which translate TL protocols to RTL and back, an existing TL model can also be used for testing an RTL component in a TL environment without the need for the whole system to be implemented in RTL already. First steps have been taken to apply Assertion Based Verification (ABV), which has been successfully used for RTL verification for years, to TLMs as well. Some of these attempts try to enhance existing approaches like SystemVerilog Assertions (SVA) [1][11] or the Property Specification Language (PSL) [8] in order to support TL designs and paradigms. In order to really use TLMs as golden reference, a full equivalence check between TLM and RTL would be desirable. One possibility to enhance the current state of the art would be to reuse existing TL assertions for the RTL design or mixed level designs. The main problem for this reuse attempt is based on the totally different synchronization methods in TLM and RTL. On RTL all synchronization is based on dedicated clock signals. In TL it is obtained by mutual dependencies of transactions and potentially by the use of time annotations in addition to the use of non-periodic trigger signals. Furthermore, the applied synchronization schemes differ on the various TL sublevels. Since a reuse of TL assertions for RTL and especially mixed level assertions has to support all synchronization schemes involved, we chose to develop our own assertion language [6][7]. As with every refinement process, e.g. synthesis, this assertion refinement requires additional information and thus can never be fully automated. A partial automation is possible by providing refinement related information upfront in order to avoid the necessity for user interaction. This automated refinement decreases time for rewriting assertions and guarantees a higher degree of consistency. In this paper we discuss which additional information is necessary for the refinement and how the process could be simplified and automated.

The paper is structured as follows. After discussing related work we give an overview of the used assertion language. Afterwards we discuss some requirements and useful features for the mixed level assertions followed by a small example. As a next step we describe a methodical refinement process from TL to RTL. We illustrate the assertion refinement by an application example which also demonstrates the usefulness of the proposed features. After a summary we give an outline of ongoing work towards packaging of assertions in a SPIRIT conformal way.

2. RELATED WORK

The application of assertion based verification to TLMs is a relatively new development. Work has been presented for migrating current RTL-ABV ap-

proaches to SystemC - the industry standard for TL modeling - as e.g. in [14], [9], and [10]. These approaches show the problem that RTL concepts cannot be directly mapped to TL, since the main synchronization mechanism for RTL designs, i.e. clock signals, normally does not exist in most TLM designs; some designs do not model timing at all. This restricts the application to the less abstract sublevels of TL or at least requires a lot more effort. In [13] a new approach for real transaction level assertions is introduced. However, transactions are mapped to signals and therefore the approach is restricted to transactions invoked by suspendable processes. Another approach is presented in [5]. Here, transactions are recorded and written into a trace in order to do post processing. As a disadvantage this approach - as every kind of trace based assertion checking - requires that everything to be recorded must be annotated in the code and the creation of simulation data bases can become very resource intensive. Furthermore this approach does not consider start and end of transactions. Therefore overlaps of transactions and parent child relations cannot be detected.

In [2] an approach for TL assertions is presented. In [3] it is shown how to apply these assertions directly to an RTL design using transactors. This approach however only allows the verification of transactions within the system while a reference to current design states is not possible. Besides, it is not meant for real mixed level assertions, since the presented assertions are all applied on the TLM side of the transactor.

In [6][7] we presented an approach working on the basis of events which covers all sublevels of TL - i.e., programmer's view (PV), programmer's view with timing (PVT), and cycle accurate (CA) - as well as RTL. We developed this language since we needed a support of both TL and RTL features as well as functionality for mixed level assertions which was not possible to the required extent with any of the existing assertion languages.

The term "Assertion Re nement" is already used in another context. In [12] an approach is presented for feeding information from the verification process back into the assertions in order to make them more suitable for the given task. Our approach for assertion refinement on the other hand deals with a transformation from TL assertions to RTL while keeping the logical relations inherent in these assertions intact.

3. TL ASSERTION LANGUAGE

In this section we give an overview of the assertion language introduced in [6][7] which provides full support for both transaction level and RTL. We show some examples of the language features.

TL models work on the basis of transactions and events. An event happens in zero time - though it might use delta delays - and can be used for triggering

threads. The language works on an event driven basis and assumes that both start and end of a transaction produce a definite event that can be used for triggering evaluation attempts of verification directives. Different levels of abstraction pose different requirements concerning the handling of events. Table 1 shows an overview of event operators and functions as well as the TL sublevels where they can be applied.

Symbol	Definition	Level
e1 | *e2*	produces an event if *e1* or *e2* occurs;	PV, PVT, CA
e1 **&** *e2*	produces an event if *e1* and *e2* occur in the same time slot;	PVT, CA
ev_expr@(*bool_expr*)	produces an event if *bool_expr* evaluates to true when *ev_expr* occurs	PV, PVT, CA
timer(*n*)	produces an event at the specified time value; **timer**(*n*) → event scheduled *n* time steps later than the current evaluation point;	PVT, CA
$delta_t	time to last event; can be used in expressions (**$delta_t** == 20)	PV, PVT, CA
*signal***'POS**	produces an event on a positive edge of the specified signal	CA
*signal***'NEG**	produces an event on a negative edge of the specified signal	CA
last_event(*event*)	evaluates to *true* if the last trigger event equals the specified event	PV, PVT, CA

Table 1. Event Operators and Functions

One of the key features of the assertion language is a general delay operator which works independently from the abstraction layer and thus allows the specification of sequences across abstraction levels. Figure 1 depicts the overall structure and functionality of this delay operator.

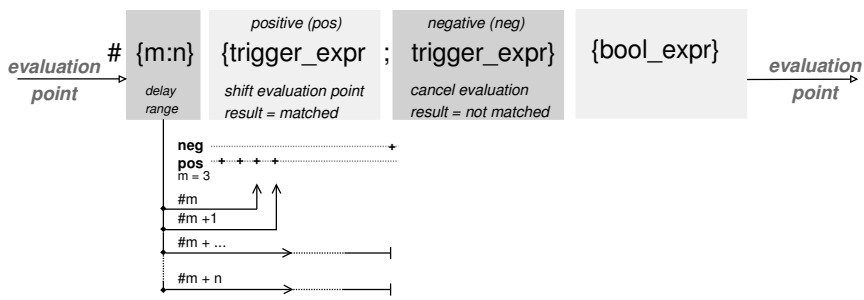


Figure 1. General Delay Operator

One evaluation attempt of a sequence built from this delay operator is called a thread[1]. The specification of a delay range leads to a split into several so-called subthreads.

Listing 1.1 shows a sample use of this delay operator. The first set of curly brackets specifies the delay range, i.e. the required number of occurrences of a trigger expression. The trigger expressions are specified within the second set of curly brackets - divided into positive triggers in front of the semicolon which shift the evaluation and negative triggers after the semicolon which stop the evaluation - while the third set of curly brackets contains the Boolean propositions to be checked at the time of the trigger.

```
#{3:5}{(e1|e2)@($delta_t>=45 && $delta_t<=50); e3, timer(51)}{A == B};
```

Listing 1.1. Sample Event Sequence

This configuration delays the evaluation until *e1* or *e2* have occurred between three and five times with a temporal distance of 45 to 50 time steps. If the positive trigger occurs three to five times, the Boolean expression ($A == B$) to the right is evaluated. If this expression evaluates to "true" the delay operator results in a "match". The delay operator results in "not matched" if either *e3* occurs, or the evaluation per delay step takes 51 time steps, or the Boolean expression evaluates to "false".

4. MIXED LEVEL ASSERTIONS

Mixed level assertions can be a great help when trying to verify an RTL component within an existing TL environment or also for checking a transactor for correct behavior.

In this section we gather some requirements for mixed level assertions and show possible applications.

Requirements

Any assertion language supporting mixed levels has to support the underlying semantics of both RTL and TL. RTL designs usually make use of dedicated clock signals in order to synchronize the different parts of the design. In contrast to that, TL designs use transaction events for synchronization purposes. For convenience reasons, the assertion language should handle clock edges and events in the same way in order to keep the modeling style for TL and RTL as similar as possible.

[1] We are using our own thread concept here, this is not a SystemC thread

Application

One common application for mixed level assertions is the validation of mixed level designs including transactors. Figure 2 shows a block diagram for two alternative setups for a simple FIFO used to synchronize a sending module (SND) with a receiving module (RCV). The FIFO can be accessed by two transactions (PUT, GET). Both transactions block the calling process in case the FIFO is full (PUT) or empty (GET). The lower part of Figure 2 shows a cross abstraction model. Here, the protocol at the receiving end is modeled with a clocked handshake and the gap between the abstraction levels is bridged by a transactor.

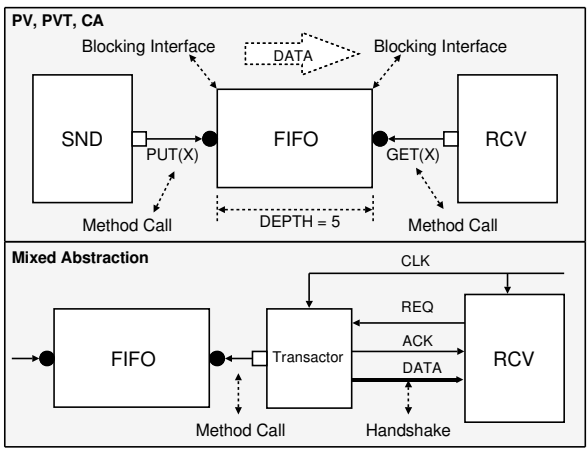


Figure 2. FIFO PUT-GET Example

Listing 1.2 shows two properties that check that data which is put into the FIFO propagates after one to six GET transactions have occurred. The upper property is working on the pure TL model whereas the lower property works on the mixed level model from Figure 2.

```
property p_DATA_PIPE_pv
 int D1;
 #1{PUT'END}{true , D1=PUT.X} |-> #{1:6}{GET'END}{GET.X == D1};
endproperty

property p_DATA_PIPE_cross
 int D1;
 #1{PUT'END}{true ,D1=PUT.X} |-> #{1:6}{ACK'POS & CLK'POS}{DATA==D1};
endproperty
```

Listing 1.2. FIFO Mixed-Level Properties

In the lower property the end of a GET transaction is captured by triggering the consequent expression only when a rising edge on both the clock and the acknowledge signal occur at the same time.

It has to be noted that in most cases mixed level assertions are written completely manually which means that the verification engineer has to know both the transaction relations involved and the underlying signal protocols. In order to ease the writing of mixed level assertions existing TL assertions could also be modified by some partial level transformations using assertion refinement as described in the section below.

5. ASSERTION REFINEMENT

Due to the very different synchronization concepts involved, an assertion refinement from TL down to RTL has to cope with several different problems: Synchronization based on events has to be transferred to one based on clocked conditions, as well as mapping the information bundled together within a transaction to a bunch of distributed registers and signals. Besides, TL designs rarely consider reset mechanisms while this is very common for RTL designs.

In this section we define our understanding of the term refinement, discuss our approach for representing transactions for RTL assertions, and introduce some new operators necessary for the refinement process.

Definition

Refinement of assertions necessarily has to follow the refinement of the design they are supposed to check. A design simulated in a simulator can basically be considered containing three different information categories:

DEFINITION 1 *Essential Information is basic information provided by the user which is derived from the original design specification, e.g. the fact that a given component shows some functionality.*

DEFINITION 2 *Modeling Information is additional information provided by the user due to the choice of the modeling implementation, e.g. the fact that the functional component uses this or that algorithm.*

DEFINITION 3 *Simulation Information is additional information brought in by the simulation semantics of the used simulation software, e.g. the fact that within this component process* A *is executed before process* B.

The first category is the only one which is completely independent from any modeling decisions made by the designer or from the choice of the software

used for testing. Hence, this kind of information[2] is also the only one which will be left definitely unchanged by refining a TL design to an RTL design. Since this information is found in both designs, a TL assertion checking this can be transformed into a similar RTL assertion by adding some RTL specific information and removing TL specific information.

Refining assertions that check information of the second category can only be refined if the verification engineer is completely familiar with both the TL and the RTL implementation, which runs against the purpose of black box testing, or if strict coding guidelines are applied to the design refinement.

Assertions checking the third category cannot be refined in general, since this would require in depth knowledge about the way the corresponding simulation software works.

DEFINITION 4 *Assertion Refinement describes a transformation of an existing assertion of a specific abstraction level to another abstraction level (usually, but not necessarily, a lower one) while ensuring that the same piece of information is checked. This transformation happens by adding required information of the new abstraction level and removing information of the old abstraction level which is no longer available.*

An example for winning information during the refinement process is the additional timing information within RTL designs which does not necessarily exist in a TL design. On the other hand the visible causal dependency between two events (e.g. *a -> b*) might disappear if they occur simultaneously in the RTL design.

Since assertion refinement does only add necessary information, all TL assertions checking essential information (see 1) can be used on all lower TL sublevels without change, i.e. a PV assertion can easily be used to verify the correctness of a PVT or CA design.

DEFINITION 5 *Partial Assertion Refinement describes the transformation of parts of an existing assertion to another abstraction level while the remaining parts are not changed.*

An example for partial refinement is the transformation of a pure TL assertion to a mixed TL-RTL assertion.

DEFINITION 6 *Illegal Assertion Refinement describes adding more information to an assertion in order to verify correlations introduced at the lower level that were not part of the upper level.*

[2]The information representation may differ, but the information itself must be available

The easiest example for that is the check of a certain timing behavior within a PVT design; the PVT level introduces timing while PV as the abstraction layer above did not contain any timing information. Thus, the original PV assertion contains less information than the derived PVT assertion and both assertions do not check the exact same behavior anymore. Since this derivation is not a legal refinement anymore, it is not discussed in the further parts of this work. Instead of illegal refinement, a completely new assertion has to be modeled.

Transaction Representation for RTL

Transactions are used for transporting data from one module to another and synchronizing their execution. A TL transaction is usually modeled as a remote function call which might be either blocking or non-blocking. On the other hand, communication in an RTL design is done via signals. The information bundled together in the transaction is usually distributed among several signals which show a certain change pattern within one or several clock cycles. Hence, each TL transaction involved has to be converted to a sequence of signal changes representing that transaction.

Here it has to be noted that it is necessary to still provide a mechanism for triggering the RTL assertions with events representing start and end of the different transactions. Otherwise, if clock edges are used instead additional information is added to the assertion which consequently does not check the same behavior as the original one anymore.

Detecting the end of a transaction is relatively easy. The simulator only has to check for the complete occurrence of the corresponding signal sequence. On the other hand detecting the start of an RTL transaction is a lot more complicated. While a TL transaction is called explicitly and can thus be easily identified from the very beginning, an RTL transaction might take several clock cycles for its complete execution; if there are more than one transaction starting with the same signal pattern in the first few clock cycles and only showing differences later on, it might be difficult to tell which transaction (if any at all) has been started by a certain signal sequence. The only possible solution to that problem lies in declaring all checked assertion results temporary until the check of the corresponding transaction sequence has been completed. At that time the result will either become valid or be discarded. The behavior of the assertion does not change in this case, only the computation becomes more resource intensive.

Parameters and return values of transactions have to be mapped to corresponding RTL registers or signals. Both these matching points and the transaction sequences have to be provided by the user.

New Operators for Assertion Refinement

The problem of still needing a representation for transaction start and end when transforming transactions to sequences can be remedied by the introduction of several new operators which can then be used as parts of the event expressions triggering a delay operator or as part of its Boolean propositions. Table 2 shows an overview of these operators.

Operator	Definition
seq'**END**	produces an event if *seq* has matched
seq'**START**	produces an event if *seq* has started
seq.**ENDED**	evaluates to "true" if *seq* has matched
seq.**STARTED**	evaluates to "true" if *seq* has started

Table 2. New Operators for Assertion refinement

These operators have to be evaluated with the occurrence of every new event, in an analogous manner to how SVA starts new evaluation threads for all assertions with the occurrence of every new clock edge.

While the detection of the end of a sequence is no problem, detecting its start is a bit more difficult. If several sequences with the same beginning signal pattern exist within the same module, a non-ambiguous detection is only possible with a certain delay as explained in the section above.

Primary and Secondary Events

If the matching of a sequence is used for triggering other sequences another problem might occur for the implementation, though: If one sequence reacts not only to the matching of another sequence but also to the event responsible for this match, it might be triggered several times instead of once.

```
sequence s1;
  #1{e1}{A == B} #1{e2}{A == C};
endsequence

sequence s2;
  #5{e2} true;
endsequence

sequence s3;
  #2{e2 | s1'END | s2'END}{B == C};
endsequence
```

Listing 1.3. Example for Primary and Secondary Events

Listing 1.3 shows an example of this situation. Sequence *s3* can be triggered by the event *e2* as well as by the match of *s1* or *s2*. In some cases the occurrence

of *e2* produces a match of one or even both sequences which might lead to an early sequence match, since one design event basically triggers both delay steps of *s3* at once. Thus, it is essential that the delay operator is only triggered once in this case.

As a solution for this we classified all events in two groups:

- Primary Events - either explicitly modeled design events or transaction events on the one hand, or external events from the timer operators used within the assertions on the other hand
- Secondary Events - derived events, in our case the start or matching of transaction sequences

If a primary event occurs during the normal execution, first all associated secondary events are computed. A specific sequence evaluation thread may be triggered at most once per primary event, regardless of the number of occurring associated secondary events.

In our example *e1* and *e2* are primary events while *s1'END* and *s2'END* are secondary events associated with *e2*. An evaluation thread for *s3* may never be triggered twice, i.e. start and complete successfully, with only one occurrence of *e2*, even if this event also causes an occurrence of *s1'END* and / or *s2'END*.

6. APPLICATION EXAMPLE FOR REFINEMENT

In this section we introduce a smart CPU subsystem and several assertions for checking its correct behavior. Using the methods and features discussed in Section 5 we show an example of TL assertion refinement.

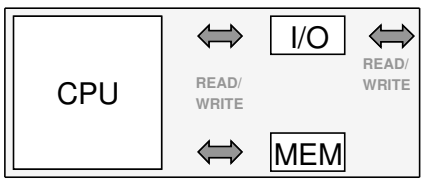


Figure 3. CPU Subsystem

Figure 3 depicts the CPU subsystem including a CPU, a memory module, and several I/O devices. All transactions in the system are modeled as remote function calls where the CPU acts as master and calls the functions provided by the memory and the I/O devices.

The correct functionality of the instruction set is checked by some existing assertions. Listing 1.4 gives an example of such an assertion. This property

```
property STD_prop
  int l_addr;
  #1{p_INSTR_WRITTEN}{p_op == cmd_STD};
  |->
  #1{read_mem'END}{true, l_addr = read_mem.param2}
  #1{write_mem'END}{(write_mem.param1 == l_addr)
                    && (p_r[p_src1] == write_mem.param2)};
endproperty
```

Listing 1.4. TL Assertion

checks the correct execution of a store instruction which first reads in the address at which to store the data and then executes the write access. The antecedent expression is triggered by writing a new instruction to the instruction register; if this instruction is a Store instruction, the implication is evaluated. The consequent first waits for the completion of a memory read access in order to get the destination address from the memory; this address is stored in the local variable *l_addr*. As a second step the completion of a memory write transaction is checked where the previously received address should be used and the data to be written has to equal the content of the specified source register *p_r[p_src1]*.

```
property STD_prop
  int l_addr;
  #1{clk'POS}{(INSTR_EN == 1) && (p_op == cmd_STD)}
  |->
  #1{clk'POS}{RD_MEM == 1, l_addr = DATA_IN}
  #2{clk'POS}{(WR_MEM == 1) && (ADDR_OUT == l_addr)
                    && (p_r[p_src1] == DATA_OUT)};
endproperty
```

Listing 1.5. Inequivalent RTL Assertion

When trying to refine these assertions without the features for detecting start and end of a transaction as listed in Table 2 the resulting assertions would look as shown in Listing 1.5. The transaction parameters have been mapped to signals and the triggering has been switched to clocked conditions. As mentioned earlier, these assertions contain additional information - in this case timing information in the form of delays by a definite number of clock cycles - and thus are not a legal refinement of the original ones. As can be seen the structure of the assertion has changed - the check for completed / started transactions has moved from the event layer to the Boolean layer and the number of delay steps has changed as well. As a consequence, as soon as the underlying RTL protocol changes, the assertions have to be adapted.

Using the operators in Table 2 on the other hand leads to the assertions depicted in Listing 1.6. The mapping of transaction parameters to signals still

takes place, but the structure of both the event trigger expressions as well as of the Boolean expressions is basically the same as that of the TL assertion.

```
sequence read_mem
  #1{clk'POS}{RD_MEM == 1}
endsequence

sequence write_mem
  #2{clk'POS}{WR_MEM == 1}
endsequence

property STD_prop
  int l_addr;
  #1{p_INSTR_WRITTEN}{(INSTR_EN == 1) && (p_op == cmd_STD)}
  |->
  #1{read_mem'END}{true, l_addr = DATA_IN}
  #1{write_mem'END}{(ADDR_OUT==l_addr) && (p_r[p_src1]==DATA_OUT)};
endproperty
```

Listing 1.6. Equivalent RTL Assertion

The additional protocol information is placed in the additional sequence declarations for the transaction sequences. A protocol change would only require an adaptation of these sequences while the actual properties remain the same. If the user provides this protocol information, the refinement process could easily be automated.

7. ACKNOWLEDGEMENT

This work has been partially funded by the European Commission under IST-2004-027580.

8. CONCLUSION AND OUTLOOK

Within this paper we gathered requirements for mixed TL-RTL assertions as well as TL to RTL assertion refinement. Further on, we suggested a methodical approach for this kind of assertion refinement which can be seen as the basis for an automated transformation from TL assertions towards RTL. We introduced the use of transaction sequences for generating the start and end event triggers as well as the concept of primary and secondary events.

An application example was given in order to demonstrate how to write mixed level assertions as well as how to apply the refinement process to given TL assertions.

Further work includes research of how to automate the refinement process by the use of meta data which guides transformation tools. This meta data might be provided manually or even obtained from existing sources.

Further useful features include partial refinement, e.g., transforming TL assertions to mixed level assertions in order to check an RTL component within

a TL system as well as the mentioned process of generating enhanced lower level assertions that check for information not available on the original higher level, e.g. generating PVT assertions with timing information from an untimed PV assertion.

Currently there are attempts of standardizing assertion APIs and generating a SPIRIT package for TL, RTL, and mixed checkers.

REFERENCES

[1] Accellera. *SystemVerilog LRM*. http://www.systemverilog.org/SystemVerilog_3.1a.pdf.

[2] N. Bombieri, A. Fedeli, and F. Fummi. On PSL Properties Re-use in SoC Design Flow Based on Transaction Level Modeling. In *6th International Workshop on Microprocessor Test and Verification (MTV)*, November 2005.

[3] N. Bombieri, F. Fummi, and G. Pravadelli. On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL. In *9th International Conference on Design, Automation and Test in Europe (DATE)*, March 2006.

[4] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. In *1st International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS)*, October 2003.

[5] X. Chen, Y. Luo, H. Hsieh, L. Bhuyan, and F. Balarin. Assertion Based Verification and Analysis of Network Processor Architectures. *Design Automation for Embedded Systems*, 2004.

[6] Wolfgang Ecker, Volkan Esen, Michael Hull, Thomas Steininger, and Michael Velten. Requirements and Concepts for Transaction Level Assertions. In *24th International Conference on Computer Design (ICCD)*, California, USA, October 2006.

[7] Wolfgang Ecker, Volkan Esen, Michael Hull, Thomas Steininger, and Michael Velten. Specification Language for Transaction Level Assertions. In *11th IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Monterey, California, Nov. 2006.

[8] H. Foster, E. Marschner, and Y. Wolfsthal. *IEEE 1850 PSL: The Next Generation*. http://www.pslsugar.org/papers/ieee1850psl-the_next_generation.pdf.

[9] A. Habibi and S. Tahar. On the extension of SystemC by SystemVerilog Assertions. In *Canadian Conference on Electrical & Computer Engineering*, volume 4, pages 1869–1872, Niagara Falls, Ontario, Canada, May 2004.

[10] A. Habibi and S. Tahar. Towards an Efficient Assertion Based Verification of SystemC Designs. In *In Proc. of the High Level Design Validation and Test Workshop*, pages 19–22, Sonoma Valley, California, USA, November 2004.

[11] IEEE Computer Society. *SystemVerilog LRM P1800*. http://www.ieee.org.

[12] Andrew Ireland. Towards Automatic Assertion Refinement for Separation Logic. In *21st International Conference on Automated Software Engineering (ASE)*, September 2006.

[13] B. Niemann and C. Haubelt. Assertion Based Verification of Transaction Level Models. In *ITG/GI/GMM Workshop*, volume 9, pages 232–236, Dresden, Germany, February 2006.

[14] T. Peng and B. Baruah. Using Assertion-based Verification Classes with SystemC Verification Library. *Synopsys Users Group, Boston*, 2003.

USING A RUNTIME MEASUREMENT DEVICE WITH MEASUREMENT-BASED WCET ANALYSIS *

Bernhard Rieder, Ingomar Wenzel, Klaus Steinhammer, and Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
1040 Wien, Austria
bernhard@vmars.tuwien.ac.at

Abstract:

The tough competition among automotive companies creates a high cost pressure on the OEMs. Combined with shorter innovation cycles, testing new safety-critical functions becomes an increasingly difficult issue [4]. In the automotive industry about 55% of breakdowns can be traced back to problems in electronic systems. About 30% of these incidents are estimated to be caused by timing problems [7]. It is necessary to develop new approaches for testing the timing behavior on embedded and real-time systems.

This work describes the integration of runtime measurements using an external measurement device into a framework for measurement-based worst-case execution time calculations. We show that especially for small platforms using an external measurement device is a reasonable way to perform execution time measurements. Such platforms can be identified by the lack of a time source, limited memory, and the lack of an external interface. The presented device uses two pins on the target to perform run-time measurements. It works cycle accurate for frequencies up to 200MHz, which should be sufficient for most embedded devices.

1. INTRODUCTION

Over the last years more and more automotive subsystems have been replaced by electronic control units (ECUs) which are interconnected by high

*This work has been supported by the FIT-IT research projects "Automatic Test Data Generation for WCET Measurements (ATDGEN)" and "Model-based Development of distributed Embedded Control Systems (MoDECS-d)".

Rieder, B., Wenzel, I., Steinhammer, K., Puschner, P, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 15–26.

dependable bus systems like FlexRay, TTP/C or CAN. Much effort has been put into increasing the reliability of communication and scheduling and great advances in these areas have been made. However, timing analysis, especially worst-case Execution Time (WCET) analysis of automotive applications, is still a challenge. This is mainly due to two factors which cumulatively increase complexity of timing analysis: More and more functionality is integrated within single ECUs [4] and the architecture of the processors is getting more complex, especially by features such as caches, pipelining, branch prediction, out of order execution and others [3]. Additionally, processor vendors do not offer accurate data sheets describing the features of their processors, so that those are unknown or have to be figured out by reverse engineering [8]. Without detailed knowledge about processor internals static timing analysis, that is calculating the time required for the execution of code without actually executing it, is often not possible.

Therefore, novel approaches use a combination of static analysis and execution time measurements to calculate a WCET bound [10]. The proposed method consists of static analysis, control flow graph decomposition, test data generation, execution-time measurement and the final calculation step. All steps are performed automatically without user interaction. The method is described in Section 3.

Computing resources of embedded applications are often limited and therefore not suitable for runtime measurements. Typical limitations are the lack of a time source, limited memory (no location to store measurement data), and the lack of an external interface (no way to transfer measurement data to host). As a solution we developed an external measurement device, which is based on an FPGA and therefore very flexible and inexpensive. We demonstrate the usage of the execution time measurement device by performing WCET calculations using a HSC12 microcontroller evaluation board. The proposed solution uses only a single instruction per measurement point and two pins for the interface to the measurement device.

Structure of this Article

This paper is structured as follows: In Section 2 we present related work in the domain of dynamic WCET estimation and execution time measurement. The measurement-based WCET analysis approach is described in Section 3. Section 4 outlines basic requirements when performing execution time measurements for timing analysis. In Section 5 we describe the hardware and firmware design of the measurement device. Section 6 explains how execution time measurements are performed by the timing analysis framework. The conducted experiments are described in Section 7. At last, Section 8 gives a short conclusion and an overview of open topics.

Contributions

The first contribution is the introduction of a dedicated runtime measurement device (RMD). The device works for CPU frequencies of up to 200 MHz. It is linked to the target using a simple 2-wire connection. Since the device is built using a field programmable gate array (FPGA) it can easily be extended or reconfigured and is very flexible. The device is especially useful to perform execution time measurements on targets with limited resources.

Second, the device is seamlessly integrated into an novel developed measurement-based WCET calculation framework as described in [10]. All operations of the measurement-based WCET analysis are performed fully automatic without user interaction.

2. RELATED WORK

Petters [5] describes a method to split an application down to "measurement blocks" and to enforce execution paths by replacing conditional jumps either by NOPs or unconditional jumps, eliminating the need for test data. The drawback is that the application is partitioned manually and that the measured, semantically changed application may differ from the real application.

Bernat et al. introduce the term of a "probabilistic hard real-time system". They combine a segmented measurement approach and static analysis, however they use test data supplied by the user. Therefore they cannot guarantee a WCET bound but only give a probabilistic estimation, based on the used test data [1].

Petters describes various ways to collect execution traces of applications in [6]. He outlines various ways how to place "observation points" and discusses benefits and drawbacks of the presented methods.

3. MEASUREMENT-BASED WCET ANALYSIS

The proposed measurement-based timing analysis (MBTA) is performed in five steps [9] as shown in Figure 1

The individual steps are explicitly described below. The measurement device hardware is used in step ④ but the description of the other steps is necessary to understand how the MBTA approach works. It is also important to mention that the current implementation is limited to acyclic code (code without loops). Since most modeling tools typically produce acyclic code this does not strongly limit the applicability of the presented method.

Static Program Analysis ①

This step is used to extract structural and semantic information from the C source code. The information is needed to perform the next steps.

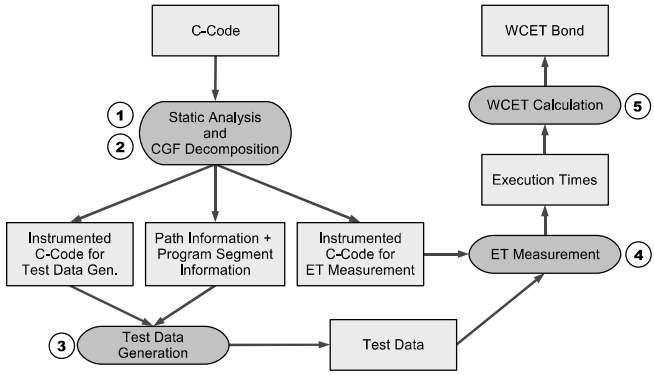


Figure 1. MBTA Framework Overview

Control Flow Graph Decomposition ②

During this phase, the control flow graph *CFG* of the analyzed program is split up into smaller program segments *PS*. The subdivision is controlled by the number of execution paths which remain within each PS, a number which is denoted as *path bound*. The command line argument for the path bound is adjustable from 1 to 2^{32} and controls the correlation between calculation time and the number of required measurements: A high path bound causes a lower number of PS and less measurement points but the sum of execution paths through all PS increases. Since each execution path of each PS has to be measured, a test data set is required for each path. Since test data calculation is very time consuming, the calculation time rises significantly [11].

Test Data Generation ③

In this step test data is calculated to execute all execution paths of each PS. The test data is acquired using a multi-stage process: Random search which is very fast is used to find most data sets. Where random search fails, model checking is used. Model checking is exhaustive, that means if there exists a data set to execute a certain path, then it will be found. If no data can be found for a particular execution path then the path is semantically infeasible. An example for semantically infeasible paths are mutual exclusive conditions in consecutive `if` statements. We used the CBMC model checker [2], which is a very fast bounded model checker that can directly process C input.

Execution Time Measurements ④

Using the generated test data all feasible paths within each PS are executed. The result of this step is an execution time profile for each PS which includes

the execution time of each path. The WCET of a given PS is the maximum of all execution time values.

WCET Calculation ⑤

The last step is the calculation of a WCET bound. The current implementation of the WCET calculation tool described in [10] uses a longest path search algorithm for a directed, acyclic graph which a single start and ending node to compute a WCET bound over all PS. This can lead to overestimations under certain circumstances, namely when the execution path with the WCET of PS_x inhibits the execution of the path featuring the WCET of PS_y. In this case WCET(PS_x) and WCET(PS_y) are accumulated in the WCET of the whole application leading to overestimation. This effect can be reduced by increasing the path bound.

4. PERFORMING EXECUTION TIME MEASUREMENTS

Execution time measurements are a precondition for the described timing analysis method. There are various methods to place instrumentation points and to perform measurements [6].

As a first method, execution traces can be made, using a cycle accurate simulator. In most cases this is not possible due to missing CPU models and documentation.

Second, pure software instrumentation can be used: A few instructions are inserted into the application that read the value of an internal time source, commonly a cycle counter located in the CPU, and write it to a output port or to a memory location. The drawback of this method is that several instructions are needed and therefore the code size and the execution time can be considerably increased.

The third option is to use pure hardware instrumentation. A logic analyzer is connected to the address bus and records the access to the memory. The advantage of this method is that no alterations on the applications are necessary. The drawback of this method is that it is expensive (the logic analyzer) and that the connection to the address bus is not always possible or the CPU has an internal instruction cache.

An interesting alternative is to use software supported hardware instrumentation. We selected this option because the modifications on the software are very lightweight (a single assembler instruction is sufficient) and the resource and execution time consumption on the target hardware is small. Since logic analyzers are often difficult to control from within an application, expensive and oversized for this task we decided to design a custom device to perform execution time measurements.

5. RUNTIME MEASUREMENT DEVICE (RMD)

The Runtime Measurement Device (RMD) acts as interface between the target and the host. It collects timestamps issued from the target and transfers them to the host. The timestamps are internally generated.

The RMD consists of a custom designed FPGA board with an Altera® Cyclone™ EP1C12C6 device which is additionally equipped with 64k of external memory (for the CPU core) and interface circuits for USB, Ethernet, RS232, RS485, ISO-K, and CAN. A modern FPGA evaluation board can also be used instead, however with a total price of approximately 300.00 Euros (with USB interface only) the custom made board is cheaper than most evaluation boards. The firmware is split up in two parts. The first part runs on a NIOS® CPU core which is clocked with 50MHz and controls the communication with the host computer. The second part is written in VHDL (Very High Speed Integrated Circuit Hardware Description Language) and operates at a clock frequency of 200MHz. This part contains the time base, which is a 32 bit counter, a FIFO to store up to 128 measurement values until they are transfered to the host and additional glue logic which recognizes measurement points and stores the actual counter value in the FIFO and synchronizes communication with the CPU core.

Since most of the design is implemented within the FPGA firmware the whole method is very flexible and can easily be adopted for custom application needs. Changes in the configuration can simply be made by uploading a new firmware design on the FPGA.

Operation

The measurement device is designed to work in two different modes. The first mode, the 2-wire mode, uses two dedicated IO pins for the measurements as depicted in Figure 1. Measurement starts when one signal drops to low. The internal counter is released and starts counting. On each measurement point, one signal drops to low, causing the counter value to be stored in the FIFO, and the other signal rises to high. If both signals are low for a adjustable amount of time, the measurement stops. According to the FIFO size up to 128 measurement points can be recorded on a single run.

The second mode is the analyzer mode. This mode is designed for very small devices. In this mode the measurement device is connected to the CPU address bus and records the time at certain predefined locations. The addresses where time tamps have to be recorded are stored in a one bit wide RAM. Measurements are taken when the output of the RAM is logical “1”. The advantage of this mode is that there need be no alterations on the target code. The disadvantages are that knowledge about the physical location of the measurement

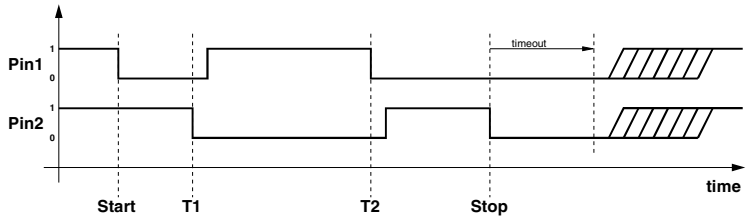


Figure 1. 2-Wire Interface Signal Waveform

points is necessary and that physical access to the address bus of the device is required, so it cannot be used for devices with on-board E(E)PROM storage.

6. INTEGRATION IN THE ANALYSIS FRAMEWORK

The measurement framework consists of a set of virtual base classes shown in Figure 2. For each virtual base class a non-virtual subclass is loaded at runtime, according to the underlying hardware.

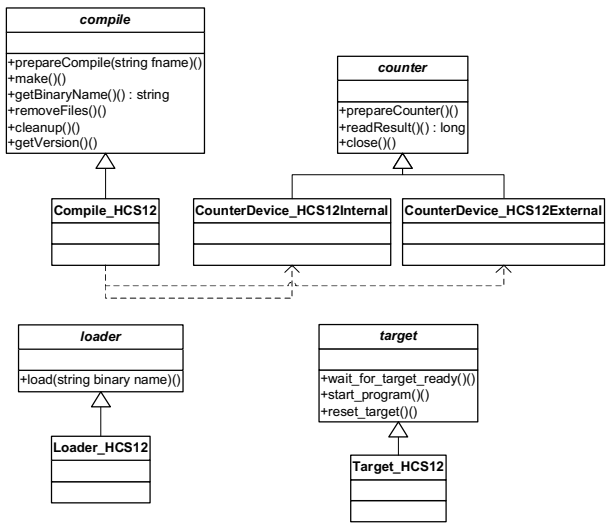


Figure 2. Measurement Application Class Framework [9]

The *compile* class is used to compile the application on the host and to generate a stub for the target to handle the communication with the host, load the test data, and to execute the application. The *counter* class activates the source code for starting and stopping the counter on the target and handles the communication between the host and the counting device for both, internal and external counting devices. The *loader* class is used to upload the application

binary onto the target, and the *target* class handles the communication between target and host from the host side.

The proposed design makes the execution time measurement application very flexible and allows an easy adoption for new target hardware. Since the measurement runs are all performed in the same way, using only a different set of subclasses, the whole framework can be adopted to support new target platforms by implementing additional subclasses.

7. EXPERIMENTAL RESULTS

We performed a series of tests using the measurement-based WCET analysis framework in combination with the internal counter device of the HCS12 microcontroller (HCS12_INTERNAL) and the runtime measurement device (HCS12_EXTERNAL). As expected, we got slightly bigger values using HCS12_INTERNAL during first test runs. This is caused by the different instrumentation methods (using the internal counter requires more instructions and therefore takes longer) and can be eliminated through calibration. To calibrate, a series of zero-length code sequences are measured and the result is subtracted from the actual measurements. For all tests the resulting WCET values were the same using both measurement methods.

The test cases we used were developed from a simple example from our regression tests (nice_partitioning_semantisch_richtig) and from automotive applications (ADCKonv, AktuatorSysCtrl, and AktuatorMotorregler). Figure 3 [9] shows the test results for all case studies for different "*Path Bounds (b)*". As mentioned before, *b* controls the maximum number of syntactically possible paths within a Program Segment (PS) after the CFG decomposition step. "*Program Segments (PS)*" shows in how many segments the Application was broken down. The next column "*Paths After Dec.*②" represents the sum of all syntactically possible paths through each PSs of the application. Interesting values are at the first and on the last line of each test case. When *b* equals 1 the total number of paths after the decomposition step equals the number of basic blocks, since one basic block comprises exactly one path. In the last line there is only a single PS and therefore the number of paths after decomposition equals the number of syntactically possible paths of the whole application.

"*Paths Heuristic*" and "*Paths MC*" describe how many paths were covered by random test data generation and how many by model checking. "*Infeasible Paths*" denotes the number of infeasible paths that were discovered during model checking. Note that these paths are also included in the number of paths covered by model checking. Infeasible paths are paths that are syntactically possible but can never be executed because the semantics of the code does not allow so. Since the number of paths covered by model checking is similar to

Application Name	Path Bound (b)	Program Segments (PS)	Paths After Dec. ②	Paths Heuristic	Paths MC	Infeasible Paths	WCET Bound [cyc]	Time MC [s]	Time ETM [s]	Overall Time [s]	Time MC / Path [s]	Time ETM / Path [s]	Paths / PS
nice_partitioning_ semantisch_richtig (46 LOC)	1	30	30	6	24	0	151	34	175	209	1.42	5.8	1.0
	5	6	14	4	10	0	151	15	39	54	1.50	2.8	2.3
	10	3	14	3	11	0	151	16	21	37	1.45	1.5	4.7
	20	2	18	2	16	3	150	22	16	38	1.38	1.1	9.0
	100	1	72	1	71	46	129	106	12	118	1.49	0.5	72.0
ADCKonv (321 LOC)	1	31	31	31	0	0	872	24	192	216	n.a.	6.2	1.0
	10	3	17	8	9	8	870	31	22	53	3.44	2.4	5.7
	100	2	74	8	66	60	872	220	17	237	3.33	1.2	37.0
	1000	1	144	12	132	132	872	483	11	494	3.66	0.9	144.0
AktuatorSysCtrl (274 LOC)	1	54	54	54	0	0	173	26	318	344	n.a.	5.9	1.0
	10	14	36	36	0	0	173	10	85	95	n.a.	2.4	2.6
	100	1	97	18	79	72	131	191	10	201	2.42	0.4	97.0
AktuatorMotorre- gler (1150 LOC)	1	171	171	165	6	6	n.a.	468	1289	1757	78.0	7.8	1.0
	10	14	92	63	29	23	3445	841	116	957	29.0	1.7	6.6
	100	7	336	57	279	247	3323	7732	62	7794	27.7	0.7	48.0
	1000	5	1455	82	1373	1325	3298	41353	49	41402	30.1	0.4	291.0

Figure 3. Test Results Of Case Studies

the number of infeasible paths (except for the first example) we see that most of the feasible paths could be found by random test data generation.

"*WCET Bound [cyc]*" gives the estimated WCET in processor cycles. To identify performance bottlenecks we did not only measure the time required for the complete WCET estimation ("*Overall Time*") but also how much of this time budget was consumed by the analysis ("*Time MC*"), which consists mainly of model checking but also includes static analysis, CFG decomposition and random test data generation, and how much time was consumed by execution time measurements ("*Time ETM*"), which consists of code generation, compilation and uploading the binary to the target.

We also observed the performance of our solution relative to the number of paths, where "*Time MC / Path*" equals $\frac{TimeMC}{PathsMC}$ and '*Time ETM / Path*" equals $\frac{TimeETM}{feasiblePaths}$. While the time required for model checking a single path within a PS is approximately constant for each test case, the time required for the execution time measurement of an individual path drops with the number of program segments. This is due to the fact that the current implementation is very simple and measures only a single PS at a time. To measure another PS the application needs to be recompiled and uploaded to the target again. On bigger

applications higher values for b should be used so the time fraction required for recompilation and uploading is much less than for a given examples. The last column ("*Paths / PS*") shows the average number of syntactically possible paths through a PS.

Regarding the path bound b it can also be noted that the quality of the estimated WCET bound improves with a rising path bound. This is caused by the fact that semantically infeasible paths are only detected by the model checker when they are located within the same PS. Therefore, the bigger the PSs the more infeasible paths can be eliminated.

The fact that the longest measurement runs took little more than 11 hours (for current industrial code) is very promising. Generating test data sets and performing the necessary measurements manually for an application of this size would take a few days at least. We think that this approach can significantly improve the testing process. It should also be noted that test data are stored and reused for later runs. For applications featuring many infeasible paths we are currently working on a solution to identify groups of infeasible paths in a single run of the model checker.

With the case study "AktuatorMotorregler" we reached the computational limit of the model checker when we set b to 1 therefore we could not get an WCET estimation value in this case.

8. CONCLUSION AND FURTHER WORK

We found that the measurement-based WCET analysis approach in combination with the presented device gives the application developer a powerful tool to perform WCET analysis. The most powerful feature is that everything is performed automatically. This saves valuable working time and eliminates cumbersome tasks like test data generation and performing manual measurements. The newly introduced measurement device makes it possible to perform run time measurements on small devices which normally would lack the appropriate hardware support (time source, memory or interface). The measurement device is cheap - and since it is based on a programmable logic device - very flexible, which allows the adaption for other hardware devices if necessary.

An important drawback is that, depending on the used RMD-target interface, at least two signal pins are required to perform measurements. Therefore the measurement device cannot be used when less than two pins are free.

The next step is to overcome limitations of the current measurement framework: loops and functions are not supported in the prototype. We are confident that these are only limitations of the current design of the prototype and of the measurement method itself. Currently a new version of the prototype which supports loops and function calls is in development.

The test data generation by model checking can be boosted by cutting off infeasible subtrees in the dominator tree. If a node cannot be reached then all other nodes dominated by it are unreachable as well. However the current implementation makes no use of this information and checks each leaf in the dominator tree which represents an unique path within a program segment.

An interesting area for improvement is the reconfiguration the RMD firmware from within the framework for different types of target hardware.

Additionally we are working on a solution to make the measurement-based approach work on more complex architectures, as those are increasingly used in new embedded solutions. The presence of caches, pipelines and out-of-order execution units impose an internal state on the processor. Different hardware states at the same instruction can result in different execution times. Therefore we are searching ways to impose a well defined hardware state while loosing a minimum on performance.

REFERENCES

[1] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. *RTSS*, 00:279, 2002.

[2] E. Clarke and D. Kroening. ANSI-C Bounded Model Checker User Manual. August 2 2006.

[3] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[4] H. Heinecke, K. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture-An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures. *Proc. Convergence, SAE-2004-21-0042*, 2004.

[5] S. Petters. Bounding the execution time of real-time tasks on modern processors. *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 498–502, 2000.

[6] S. M. Petters. Comparison of trace generation methods for measurement based WCET analysis. In *3nd Intl. Workshop on Worst Case Execution Time Analysis*, Porto, Portugal, July 1 2003. Satellite Workshop of the 15th Euromicro Conference on Real-Time Systems.

[7] Rapita Systems. Rapitime whitepaper. 2005.

[8] C. Thomborson and Y. Yu. MEASURING DATA CACHE AND TLB PARAMETERS UNDER LINUX. *Proceedings of the 2000 Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 383–390, 2000.

[9] I. Wenzel. *Measurement-Based Timing Analysis of Superscalar Processors*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[10] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based worst-case execution time analysis. *Software Technologies for Future Embedded and Ubiquitous Systems, 2005. SEUS 2005. Third IEEE Workshop on*, pages 7–10, 2005.

[11] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Proc. Conference on Design, Automation, and Test in Europe*, Mar. 2005.

IMPLEMENTING REAL-TIME ALGORITHMS BY USING THE AAA PROTOTYPING METHODOLOGY

Pierre Niang, Thierry Grandpierre, Mohamed Akil
ESIEE Paris, Lab. A^{2}SI, Cite Descarte BP 99, Noisy Le Grand 93160, Cedex France
niangp@esiee.fr; t.grandpierre@esiee.fr; akilm@esiee.fr

Abstract: This paper presents a system-level methodology (AAA) for signal and image processing algorithms onto circuit architecture. This AAA (Algorithm Architecture Adequation) methodology is added and implemented in an existing software dedicated to the fast prototyping and optimization of real-time embedded algorithms onto multicomponent architectures. The resulting tool, called SynDEx-IC, is a free graphical Computer-Aided Design (CAD) software. It supports different features: algorithm and architecture specifications, data path and control path synthesis, performances prediction, optimizations and RTL generation.

1. INTRODUCTION

Digital signal processing applications, including image processing algorithms, require growing computational power especially when they are executed under real-time constraints. This power may be achieved by the high performance of multicomponent architectures based on programmable components (processors) which offer flexibility and non programmable components (reconfigurable circuits) which offer higher performances with less flexibilities. Consequently, there is a need for dedicated high level design methodology, associated to efficient software environments to help the real-time application designer to solve the specification, validation optimization and synthesis problems. Several research efforts have addressed the issue of design space exploration and performance analysis of the embedded systems. Therefore, some methodologies and tools have been developed to help designers for the implementation process. Among these different researches for multicomponent designs, one may cite the SPADE methodology [LWVD01] and the CODEF tool [ACCG01]. Although these tools are very efficient, none of them is able to bring together under real-time and resources constraints: unified models,

Niang, P., Grandpierre, T., Akil, M, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 27–36.

graphical specification, performances prediction, generation of distributed and optimized executives for programmable part and generation of optimized RTL code for configurable part. Actually, the tool associated to AAA methodology [GS03] is named SynDEx. It supports mainly multiprocessor architectures but does not allow to address the optimization and VHDL generation process of configurable part. Hence, there is a need to extend the AAA for circuit and to develop an associated tool. The presented work is an intermediate step for tending towards a codesign tool associated to the AAA methodology. There are several tools allowing to automate the optimized hardware implementation for reconfigurable circuits from a high-level design specification. One may cite Esterel Studio tool, where the user captures a design specification and then automatically generates the hardware description in HDL-RTL. Another high-level synthesis framework, SPARK [GDGN03], provides a number of code transformations techniques. SPARK takes behavioral C code as input and generates Register Transfer Logic (RTL) code. This RTL code may be synthesized and mapped onto an FPGA. One may also cite Simulink which is an extension of MATLAB that allows to create algorithm in a graphical fashion. This allows the engineer to visually follow algorithms without getting lost in low level code. Simulink uses a System Generator which takes this graphical algorithmic approach and extends it to FPGA development by using special Simulink blocks. However, none of them is integrated or interfaced with a codesign tool for the multicomponent implementations. The remainder of this paper is centered on the AAA/SynDEx-IC and the implementation of image processing applications by using SynDEx-IC tool. In Section 2, the transformation flow used by AAA methodology is introduced. In Section 3 the software tool SynDEx-IC which implements the AAA methodology for circuits is presented. Section 4 introduces the implementation of image processing onto an FPGA. Finally, Section 5 concludes and discusses the future work.

2. AAA METHODOLOGY FOR INTEGRATED CIRCUITS

AAA is supported by SynDEx tool which is based on dedicated heuristics for the distribution and scheduling of a given algorithm onto programmable components. SynDEx uses graph theory in order to model multiprocessor architectures, applicative algorithms, the optimization and code generation. We will extend the AAA methodology for integrated circuits. In the case where the implementation does not satisfy the constraints specified by the user, we apply an optimization process in order to reduce the latency by increasing the number of circuit resources used.

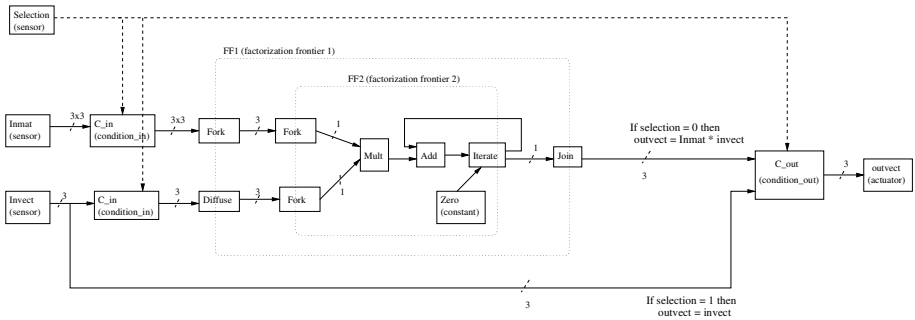


Figure 1. Algorithm graph of CMVP

Algorithm specification

The algorithm is modelled by an oriented hyper-graph G_{al} of operations (graph vertices O), its execution is partially ordered by their data-dependences (oriented graph edges D with $D, G_{al} = (O, D)$). In the algorithm graph, each vertex represents a computation operation, an input/output operation, a conditioned operation (if then else) or a finite factorization frontier for repetitive subgraphs (for i=x to y): a finite factorization frontier is an abstraction delimited by factorization frontier vertices (Fork, Diffuse, Join, Iterate), an infinite factorization frontier is the interface with the external environment delimited by input/output operation. This data-dependence graph, also called directed acyclic graph (DAG), exhibits a potential parallelism. It is important to note that the factorization serves for simplifying the algorithm specification and it is used to explore different optimized implementations. It does not involve a sequential execution inevitably (i.e a "loop") in the circuit. To illustrate the algorithm model, the figure 1 presents a Conditioning Matrix-Vector Product (CMVP). If "selection" input is equal to 1 then "outvect" output copy the "invect" input else if "selection" input is equal to 0 then we have the Matrix-Vector ("inmat" and "invect") product on the "outvect" output. The choice of this example was motivated by regular computations on different array data which highlight the use of the factorization process.

High level synthesis based upon graph transformations

The hardware implementation of the factorized data dependence graph consists of providing a corresponding implementation for the data path and the control path of the algorithm. In this part of the paper, the high level synthesis of the control path is introduced [LAGS04]. The control path corresponds to the logic functions to add to the data path, in order to control the multiplexers and the transitions of the registers composing the modules perform-

ing the graph operations. The control path is then obtained for synchronization of data transfer between registers. In order to synthesis this control path, SynDEx-IC starts by building the neighborhood graph corresponding of the algorithm graph. The neighborhood graph is an interconnections graph of the factorization frontiers, the sequential operations (operation driven by a clock), the operations specific to conditioning (Condition_In, Condition_Out) and the combinative operations (operation whose response at the moment t depends only on its input at the moment t) . According to the data dependences relating theses vertices, every vertex may be a consumer or/and producer relatively to another vertex. The control path graph is composed of CU vertices (Control Unit) connected between them according to the connections relations of the neighborhood graph. Moreover, if the consumer data comes from various producers with different propagation time, it is necessary to use a synchronized data transfer process. This synchronization is possible through the use of a request/acknowledge communication protocol. Consequently, the synchronization of the circuit implementing the whole algorithm is reduced to the synchronization of the request/acknowledge signals of the set of CU vertices. In the control path graph, we have four kinds of CU vertices: control unit of factorization frontier, control unit of a sequential operation, control unit of a combinative operation block delay and control unit of a conditioning operation.

Optimization of the implementation

The target architecture is made of one reconfigurable circuit (FPGA). Once specified the algorithm and architecture, it is necessary to characterize each operation of the algorithm graph in order to be able to perform the implementation optimization process. In this context, designers have to specify the latency time and the logical units number occupied on the FPGA by each operation of the algorithm graph. This information can be obtained starting from traditional tools for synthesis. A heuristic was developed and implemented to check an adequate implementation for the available circuit while satisfying the con-

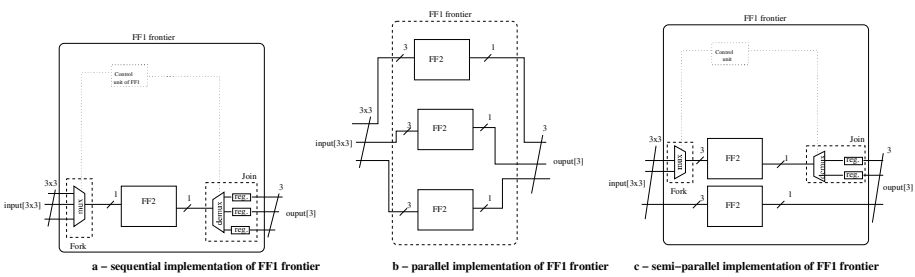


Figure 2. Three examples of implementation of FF1 frontier of the CMVP

straint latency which be specified by designers. For each repeated hierarchical operation, there are several possible implementations. On a FPGA, the CMVP can be implemented in a purely sequential way (figure 2-a), in a purely parallel way (figure 2-b) or in a semi-parallel way i.e. mixing the sequential and parallel implementation (figure 2-c). The sequential implementations require the synthesis of a dedicated control path (constituted of multiplexers, demultiplexers, registers and control units). Among these three implementations, the purely sequential implementation (a) uses less surface (number of logical units occupied by the set of algorithm graph operations) but is slower. The purely parallel implementation (b) is faster but occupies the maximum of surface on the FPGA. The implementation semi-parallel (c) is an example of compromise between the surface occupied by the algorithm and the latency time of this algorithm, it generates an execution time twice faster than that of (a) and requires also about twice more surface. The combination of all the possible implementations of the repeated operations constitutes the implementations space which have to be explored in order to find a optimized solution. Consequently, for a given algorithm graph, there is a great number of possible implementations. Among all these implementations, it will be necessary to choose one which satisfies the real-time latency constraint and which fit into the FPGA. This is why we developed heuristic ones guided by a cost function. The cost function allows to compare only a part of the solutions which seems most interesting of this implementations space. Thus, this heuristic is based on an greedy algorithm rapid and effective whose cost function f is based over the critical path length of the implementation graph: it takes into account the latency of the application T and the implementation surface A which are estimated (performance prediction) by SynDEx-IC starting from the characterization.

3. SYNDEX-IC: A SOFTWARE FRAMEWORK

SynDEx-IC tool is developed starting from the version 6.6.1 of SynDEx. The coding was performed in CAML language as for SynDEx. The design flow of SynDEx-IC presented in figure 3 is made up of three parts which will be introduced below.

Applications specification

This part is the starting point of SynDEx-IC design flow (Cf part A of the figure 3). The figure 4 details the graphical interface for the algorithm specification: each box represents either a computation operation (“Add”, “Mul”) or an input/output operation (“input”, “output”) which is used to transmit data of an iteration to another. Then, it is necessary to specify: the algorithm latency constraint (execution time of all the application) and the features of the target FPGA (for each element of the algorithm graph, it is necessary to define its

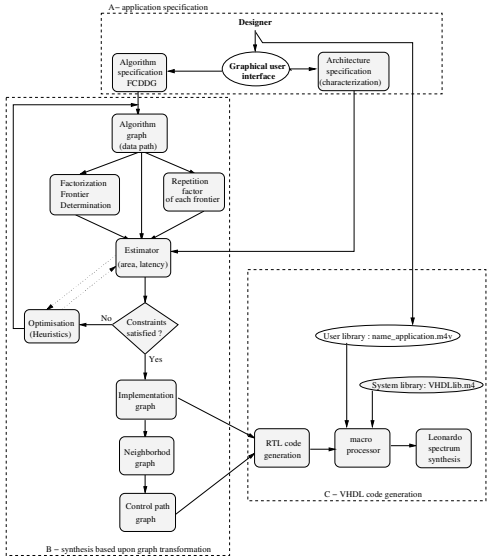


Figure 3. The AAA/SynDEx-IC design flow

latency and the quantity of resources necessary to its material implementation on the target component). Besides, the user needs to specify the performing frequency of all the sequential operations used in the algorithm.

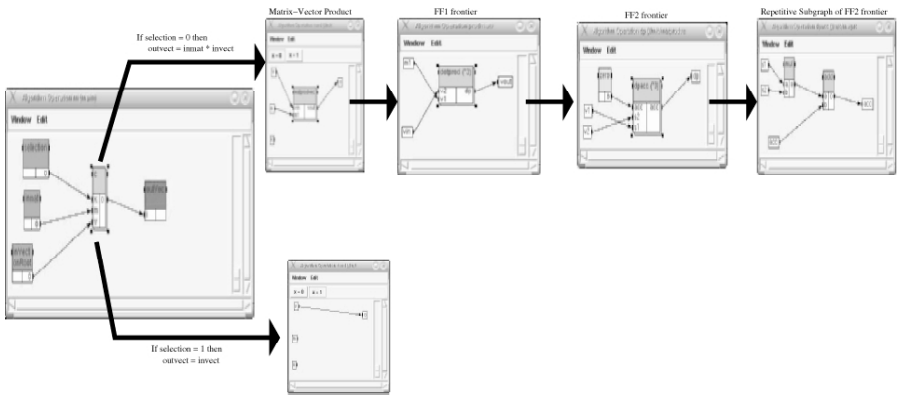


Figure 4. SynDEx-IC snapshot of a CMVP

Graph transformation and optimization

This section is the core of SynDEx-IC design flow (Cf part B of the figure 3). From the algorithm graph (data path) of the application, the graph transformation of all the graph is proceeded The function of graph transformation allows the insertion of the specific operations which make it possible to mark the repetition and of the conditioning operations: Fork, Join, Iterate, Diffuse, Condition_In, Condition_Out. Thus, the determination of the frontiers and of the factorization ratio of each frontier is processed before to begin the optimization heuristic. It seeks an optimized implementation graph of the application which satisfies the latency constraint. This implementation graph is used to build the neighborhood graph in order to synthetize the control path.

Automatic VHDL code generation

In order to implement the application algorithm onto the corresponding circuit we need to generate the data path as well as the control path. This code generation is done by way of an intermediate macro code which is used to have a code generator independent of the hardware description language Each operation of the optimized implementation graph corresponds to a RTL module which will be implemented on a reconfigurable component.

4. FPGA IMPLEMENTATION OF THE PROCESSING ALGORITHMS

In order to illustrate the use of SynDEx-IC tool, real time algorithms in image processing were implemented onto Xilinx FPGA (spartan XC2S100). The aim of this project is to carry out an application of video processing using an electronic card based on FPGA. Algorithms of video processing were implemented as well as an electronic card for interfacing the FPGA with a video camera and a monitor. The FPGA treating only digital signals, it is necessary to digitize the composite video signal. Thus, one will use an analogical/digital converter (ADC) so that the FPGA can process the pixels, as well as a digital/analogical converter (DAC) to restore an analogical signal at exit. Moreover to process a video image, it is necessary to use the synchronization signals of the video. For that, one will use an extractor of synchronization.

Implementation onto FPGA using SynDEx-IC software

Several filters of image process (robert, prewitt, sobel, canny-deriche, low pass) were developed. However, in this part of the paper only sobel and prewitt filters (contours extraction) will be introduced for lack of place. In order to implement these filters, one started by specifying the data-flow graph of the application, the latency constraint and then the characterization of the target com-

ponent. Thus, to obtain optimized VHDL code of the application, SynDEx-IC performs the heuristic of optimization. The algorithmic specification of filters may be performed from the transform in Z of the transfer function. This specification can also be made starting from the application of a convolution matrix. The transfer function of the sobel filter is divided in two parts: the horizontal

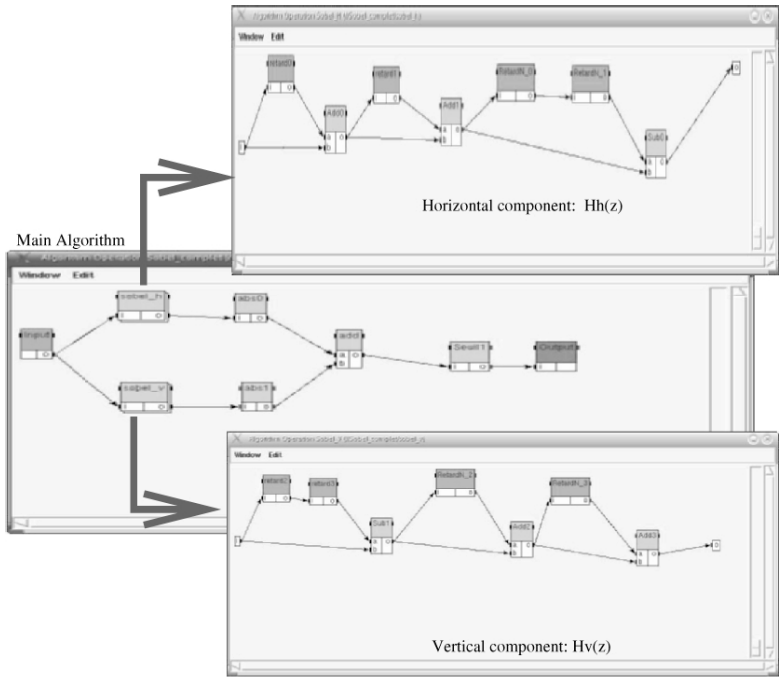


Figure 5. snapshot of sobel filter

component and the vertical component. Thus, the transfer function of the horizontal component is presented as follows: $Hh(z) = (1 + z^{-1})^2(1 - z^{-2N})$ while the vertical component: $Hv(z) = (1 - z^{-2})(1 + z^{-N})^2$
The figure 5 is a SynDEx-IC specification of the sobel filter using the transfer function. In this specification, the initial image is treated by each component (horizontal and vertical) of the filters, then the absolute values ("abs") of the pixels of the two resulting images are summoned two to two (the calculation approximation of the gradient result amplitude: modulate of the two images resulting). Lastly, we have a function of threshold ("seuil") which makes it possible to lower the level of gray of the image result. The sensor "input" represents the input pixels while the actuator "output" represents the output pixels. This algorithmic specification needs also two storage operations: a component "retard" allowing to store a pixel during a signal of clock and a component "retardN" allowing to store a pixel throughout all a line.

Filters Filters	Number of CLB	Number of Flip Flop	Number of Ram block	critical time (ns)
Sobel (transfer function)	376 of 1200	397 of 2400	8 of 10	36.066
Sobel (hand filter)	334 of 1200	356 of 2400	8 of 10	31.174
Prewitt (transfer function)	579 of 1200	424 of 2400	10 of 10	35.946
Prewitt (hand filter)	525 of 1200	386 of 2400	10 of 10	29.572

Table 1. comparison table of the filters synthesis

Implementation results

Starting from code generated by SynDEx-IC, one simulated the filters of sobel and prewitt by using the modelsim tool. Thus, one obtained images results enough interesting for various input images. If one compares the results obtained for code generated by SynDEx-IC with the results obtained by coding these filters in language C, one noted that they were identical. Thus, for somebody who knows neither the VHDL, nor the language C it may be more practical to use graphical specification SynDEx-IC to design these filters of image processing. Once the simulation of generated code made, one proceeded to the implementing of these filters on the Spartan by using the xilinx webpack tool. These logic synthesis results seen in the TABLE 1 show that compared to the hand filters (make by a manual designer), the SynDEx-IC filters are less interesting but easier and more rapid to design for the user. A reduction area estimated to about 10% and a increasing delay time estimated to about 15% are achieved for hand compared to SynDEx-IC.

5. CONCLUSION AND OUTLOOK

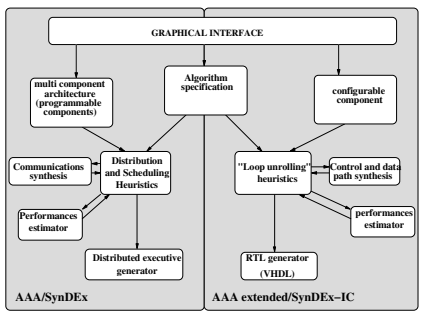


Figure 6. AAA methodology and its extension

Given a single data-flow graph specification, it is possible to generate a multiprocessor optimized implementation using SynDEx or an FPGA optimized implementation using SynDEx-IC. Moreover, the development flow is unified from the application designer point of view (Cf figure 6). We have presented the different steps to generate a complete RTL design corresponding to the optimized implementation of an application specified on SynDEx-IC. This work is an intermediate step in order to finally provide a methodology allowing to automate the optimized hardware/software implementation. The next step will be to merge SynDEx-IC and SynDEx in order to support **mixed** parallel architectures (architectures with programmable components **and** reconfigurable circuit). To support mixed parallel architectures, the partitioning problem between programmable and configurable components should be resolved in first. For that purpose, it will be necessary to connect SynDEx and SynDEx-IC heuristics. We are developing new hardware/software partitioning heuristics for programmable and configurable components. Thus, the automatic communication synthesis between these two parts is being studied. This work is supported by the Conseil Régional d'Ile de France (PRIMPROC Project).

REFERENCES

[ACCG01] M. Auguin, L. Capella, F. Cuesta, and E. Gresset. Codef: A system level design space exploration tool. In *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, page 4, Salt Lake City, USA, may 2001.

[GDGN03] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark, high-level synthesis framework for applying parallelizing compiler transformations. In *Intl. Conf. on VLSI Design*, Mumbai, India, January 2003.

[GS03] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *First ACM & IEEE Intl. Conf. on formal methods and models for codesign. MEMOCODE'03*, Mt Saint-Michel, France, june 2003.

[LAGS04] L.Kaouane, M. Akil, T. Grandpierre, and Y. Sorel. A methodology to implement real-time applications on reconfigurable circuits. In *Special issue on Engin. of Config. Systems of the Journal of Supercomputing*. Kluwer Academic, 2004.

[LWVD01] P. Lieverse, P. Van Der Wolf, K. Vissers, and E. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. In Journal of VLSI Signal Processing Systems, editor, *Special issue on signal processing systems design and implementation*, volume 29, page 10, Hingham, MA, USA, November 2001. Kluwer Academic Publishers.

RUN-TIME EFFICIENT FEASIBILITY ANALYSIS OF UNI-PROCESSOR SYSTEMS WITH STATIC PRIORITIES

Karsten Albers, Frank Bodmann and Frank Slomka
Department for Embedded Systems/Real-Time Systems, University of Ulm
{name.surname}@informatik.uni-ulm.de

Abstract: The performance of feasibility tests is crucial in many applications. When using feasibility tests online only a limited amount of analysis time is available. Run-time efficiency is also needed for testing the feasibility of many different task sets, which is the case in system synthesis tools. We propose a fast uni-processor feasibility test using static priorities. The idea is to use approximation with a variable error to achieve a high performance exact test. It generally outperforms the existing tests which we show using a large number of random task sets.

Key words: Real-Time Systems, Fixed-Priority Scheduling, Feasibility Analysis, FPTAS.

1. INTRODUCTION

The performance of the real-time analysis is important for the automation of the design process for embedded systems. Three different kinds of feasibility tests exist: Exact tests, sufficient tests and approximations. The most used exact test is the worst case response time analysis by Audsley et al. (1993). Sjödin and Hansson (1998) propose an efficient implementation of this analysis. The exact test proposed by Manabe and Aoyagi (1998) has exponential time complexity and is outperformed by the worst case response time analysis in most cases. The sufficient test by Liu and Layland (1973) and by Bini et al. (2001) are designed for RM scheduling and therefore not

The research described has been supported by the Deutsche Forschungsgemeinschaft under grants SL47/1-1 and SL47/2-1

Albers, K., Bodmann, F., Slomka, F, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 37–46.

suitable for more general models. Approximations allow a trade-off between run-time and acceptance rate. A predefined error allows a certain amount of inexactness while performing the test which leads to a speed-up for the tests. Albers and Slomka (2004) gives such an approach for earliest deadline first (EDF) scheduling and extended it to a fast exact schedulability analysis (Albers, Slomka 2005). It takes advantage of the approximation approaches and allows to dynamically adjust the error during the schedulability analysis. Fisher and Baruah (2005a, 2005b) transferred this approximation to systems with static priorities. The goal of this paper is to extend the results achieved with the exact dynamic test to systems with fixed priorities.

2. MODEL

We consider the sporadic task model on uniprocessor systems using pre-emptive fixed priority scheduling. The results can be extended to more advanced task models. Each task τ_i of a task set $\Gamma=\{\tau_1,\ldots,\tau_n\}$ is described by an initial release time (or phase) ϕ_i, a relative deadline D_i (measured from the release time), a worst-case execution time C_i (cost) and a minimal distance (or period) T_i between two instances of a task. The priority assignment of the tasks follows the deadline monotonic approach. The task wih the smallest deadline gets the highest priority. An invocation of a task is called a job, and the k^{th} invocation of each task τ_i is denoted $\tau_{i,k}$. The release time $\phi_{i,k}$ of $\tau_{i,k}$ can then be calculated by $\phi_{i,k} = \phi_i + (k-1)\cdot T_i$, and the deadline $d_{i,k}$ by $d_{i,k} = \phi_i + D_i + (k-1)\cdot T_i$. In the following the synchronous case is assumed, so $\forall \tau_i \in \Gamma : \phi_i = 0$. This leads also to a sufficient test for the asynchronous case. One feasibility test for this kind of task sets is proposed in by Baruah (2003) using the concept of demand and request bound functions. The idea is to define functions calculating the maximum sum of execution times of all jobs within an interval of length *I*. The request bound function considers all jobs that can arrive within a time interval of the length *I*. For the demand bound function it is additionally necessary that the deadline of all jobs occurs within *I*. To get the maximum demand bound contribution of a task τ_i it is necessary to calculate the sum of costs of all *k* consecutive jobs of the tasks for which the distance of the deadline of the *k*-th job of the task τ_i to the release time of the first job of the task τ_i is smaller than the length of the interval *I*: $d_{i,k} \le I$. *The maximum cumulated execution requirement of all jobs $\tau_{i,k}$ with $\tau_i \in \Gamma$ having request time and deadline within I* is called demand bound function:

$$dbf(I,\Gamma) = \sum_{\forall \tau_i \in \Gamma \wedge I \ge D_i} \left\lfloor \frac{I - D_i}{T_i} + 1 \right\rfloor \cdot C_i$$

Let Γ_τ be the task set that contains only task τ. We define $\mathrm{dbf}(I, \tau) = \mathrm{dbf}(I, \Gamma_\tau)$ as a shortcut description. Baruah also defines the request bound function: *The maximum cumulated execution requirement of all jobs $\tau_{i,k}$ with $\tau_i \in \Gamma$ which have their release times within I*

$$\mathrm{rbf}(I, \Gamma) = \sum_{\forall \tau_i \in \Gamma} \left\lceil \frac{I}{T_i} \right\rceil \cdot C_i$$

Same as above we also use $\mathrm{rbf}(I, \tau) = \mathrm{rbf}(I, \Gamma_\tau)$ as a shortcut description. For an interval *I* and a task τ_i a set of consecutive jobs of the task contributes fully to the request bound function of *I*, if the difference between the release time of the first job of the set and the release time of the last job of the set is equal or smaller than the length of *I*. For a feasibility test only intervals that start at an idle point of the system are of interest (i. e. a point in time where no request is ready for execution). The synchronous release of all tasks τ_i is taken as the start point of all intervals. For systems using EDF scheduling the following condition leads to a necessary and sufficient test: $\forall I > 0 : \mathrm{dbf}(I, \Gamma) \leq I$ Based on this condition Baruah (2003) gives a schedulability analysis and Albers and Slomka (2004) an approximation. For static priority scheduling, it is not sufficient to test the demand bound function. Instead it is necessary to consider for each task the interruption of higher priority tasks and therefore each task separately. Let $\Gamma_{hp(\tau)}$ be the task set which contains all task with a priority higher than that of task τ. A feasibility test for such a system is given by Baruah (2003): *A task system is feasible in regard to task τ if and only if for each absolute deadline of a job $d_{\tau,k}$, $k \in N$ there exist an interval $I' \leq d_{\tau,k}$ (and $I' \geq d_{\tau,k}-T_\tau$) for which the following condition holds:* $\mathrm{dbf}(d_{\tau,k}, \tau) + \mathrm{rbf}(I', \Gamma_{hp(\tau)}) \leq d_{\tau,k}$.

A job can satisfy its deadline if there exists an interval *I*, with length equal or smaller than the deadline of the task, in which the system has an idle point with respect only to the regarded jobs of this task and all jobs of higher priority tasks. One of these intervals is the response time of the task in question. Baruah proposes to first check the inequation above for $I' \leq d_{\tau,k}$ and than calculate the response time if necessary. To keep the test tractable the number of tested jobs of τ_i needs to be limited by an upper test border. Baruah has shown such a border for the recurring real-time task model.

3. APPROXIMATIONS

As a prerequisite for achieving an exact test with a good performance, it is necessary to develop a new approximation for static priority scheduling. In contrary to the previously know approximation we need an algorithm in which the contributions of the tasks are calculated in an incremental way. To

get such a selectable sufficiency the error which occurs due to the approximation needs to be bounded.

3.1 Static feasibility test

The idea of the approximation is to define a cumulated request bound function*: The cumulated request with respect to task τ is the request of all jobs of all tasks with a higher priority than τ which can arrive within an interval I and the demand of $\tau$for this interval:*

$$\mathrm{Cumu}(I, \tau) = \mathrm{rbf}(I, \Gamma_{hp(\tau)}) + \mathrm{dbf}(I, \tau)$$

Let us now consider the cumulated request for those intervals for which the condition $I' = d_{next}(I', \tau)$ holds. *The additional request for interval I is the difference between the cumulated request and the interval:* $\mathrm{addReq}(\tau) = \mathrm{Cumu}(I, \tau) - I$. *The part of the jobs that cannot be processed within I due to the arrival times of the jobs is called exceeding costs.*

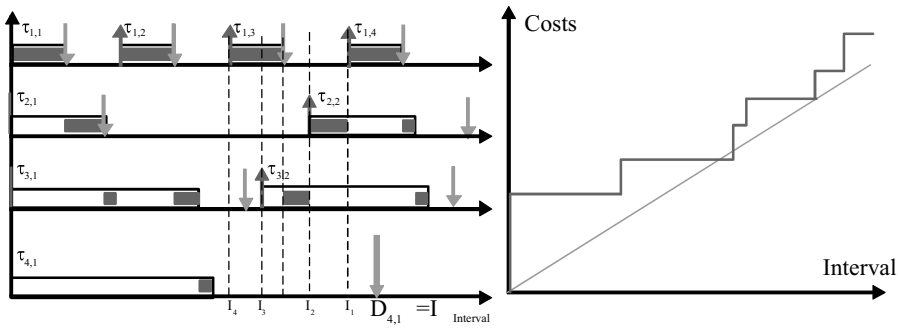


Figure 1. a) Schedule for example, b) Cumulated request for schedule

There are two possible situations in which exceeding costs can occur. In the first one the arrival of a job occurs so late that the remaining time to the end of *I* is shorter than the execution time necessary for the job. It can than never be completed within *I*. Consider the example task set $\tau_1 := (C_1 = 4, D_1 = 4, T_1 = 8)$, $\tau_2 := (C_2 = 3, D_2 = 7, T_2 = 22)$, $\tau_3 := (C_3 = 3, D_3 = 17, T_3 = 19)$, $\tau_4 := (C_4 = 1, D_4 = 26, T_4 = 30)$. The schedule can be found in Fig 1a and the cumulated costs function belonging to it in Fig 1b. The job $\tau_{1,4}$ belongs to the first situation. The difference between its release time ($\varphi_{1,4}=24$) and *I* ($d_{4,1}=26$) is smaller than its execution time. One part of this job (length 2) has to be scheduled outside of *I* and therefore counts to the exceeding costs. The second kind of situation occurs when the task is preempted and therefore its execution is delayed so that at least a part of the execution has to occur after the end of *I*. One example for this situation is the job $\tau_{2,2}$ in Fig 1. Due to its preemption by $\tau_{1,4}$ it is not possible to process $\tau_{2,2}$ completely within *I*, therefore it partly counts to the exceeding costs. Same for $\tau_{3,2}$ which is delayed and preempted by three other jobs ($\tau_{1,3}$, $\tau_{2,2}$, $\tau_{1,4}$). All tasks with a priority higher than the priority of τ can contribute to the

exceeding costs and it is not relevant to know which task exactly contributes how much: Lemma: *The deadline of a job $\tau_{i,k}$ is met if and only if either for I = $d_{i,k}$: the cumulated request for I is smaller or equal to I (cumu(I, τ_i)≤I) or the additional request of I is covered by exceeding costs.* Proof: Suppose that the deadline of a job τ_i is not met despite that the cumulated request does not exceed the sum of *I* and the exceeding costs. Than costs of at least one job do not fit within *I* and the execceding costs. This is only possible if either there exists an idle time in *I* (with respect to τ_i and all tasks with a higher priority) or a job is processed within *I* that is not included in *cumu*. In case of the idle time $\tau_{i,k}$ has also been processed completly before the end of its deadline. The second case is that a job of a higher priority task, which has not been requested within *I* is processed within *I*. *I* starts at an idle point of the system (by definition) and processing a task with a request time outside of *I* would be in contradiction to the definition. The other possibility would be to process a job $\tau_{i,m}$ with $m>k$. But this can only be done if the considered job $\tau_{i,k}$ is already finished, so the deadline $d_{i,k}$ holds.

The idea for an implementation is to start with the additional request and try to reduce it step by step by considering the exceeding costs. If they cover the additional request the test succeeds otherwise the test fails.

3.2 Approximation with bounded deadlines

Using the proposed algorithm an approximation can be achieved. The idea is to limit the number of test points for the cumulated request function. For each task only a limited number of test points (one for each of the first *k* jobs) are considered exactly. The remaining test points are skipped and an approximation is used instead. In the example in Fig 2 the first two jobs of τ_1 are considered exactly, the remaing are approximated using the specific utilization of the task. *An upper bound for the task request bound function considering only the first k jobs exactly is called upper approximated task request bound function:*

$$rbf_{sup}'(I, \Gamma_i, k) = \begin{cases} rbf(k \cdot T_i, \Gamma_i) + \frac{C_i}{T_i} \cdot (I - d_{i,k}) & I > k \cdot T_i \\ rbf(I, \Gamma_i) & I \le k \cdot T_i \end{cases}$$

$$rbf_{sup}(I, \Gamma, k) = \sum_{\forall \tau_i \in \Gamma} rbf_{sup}'(I, \Gamma_i, k)$$

The error of the approximation is bounded as shown in Fig 2. The difference between the approximated and the real request bound function of task τ_i is bounded by one times C_i. In the approximated part of the function the value of rbf_{sup} is at least $k*C_i$. So, as *rbf* is a non-decreasing function the relative error ε is limited to $1/k$.

$$\varepsilon = \frac{\mathrm{cumu}_{sup}(I, \Gamma_i, k) - \mathrm{cumu}(I, \Gamma_i)}{\mathrm{cumu}(I, \Gamma_i)} \leq \frac{1}{k}$$

This also limits the overall error to *1/k*. *k* and therefore the degree of exactness can be chosen. The complexity of this approach can be calculated by considering that this test has to be done for each task separately. For a task τ_i with $D_i \leq T_i$ only the first job of the task has to be considered. The number of test points (of the approximated cumulated request bound function) is limited to *k*n*, where n is the number of tasks with a higher priority than τ_i. This leads to an overall complexity $O(\log n \cdot n^2 \cdot 1/\varepsilon)$.

3.3 Approximation with arbitrary deadlines

The proposed approximation can be extended to the case of arbitrary deadlines ($D_i > T_i$). The question is which jobs of τ_i are necessary to be tested. For the periodic general task model, it was sufficient to test only the first job. For arbitrary task systems this is not sufficient any more because several jobs with the same priority can exist concurrently. Baruah has shown a border for the recurring real-time task system. The idea is to find the point of time where it is guaranteed that the cumulated request bound function does not exceed the intersection any more. *There exists a point of time $t_0 \geq 0$ such that for all $t' \geq t_0$: $cumu(t', \tau) \leq t'$*. Using the approximated cumulated request function such a point is guaranteed to exist using a task system with an overall utilization lower than 1. Starting at the point where the *rbf* for the last task switch to approximation, the remaining function is monotonically increasing with a slope equal to the utilization of the function. Therefore at one point this function must fall below the intersection. The overall number of test intervals for each task is limited to *k*. This is the case even if several instances of a task are considered. Therefore the overall number of test intervals is limited to *n·k* where n is the number of tasks. Therefore the complexity of this test is the same as in the non-general case.

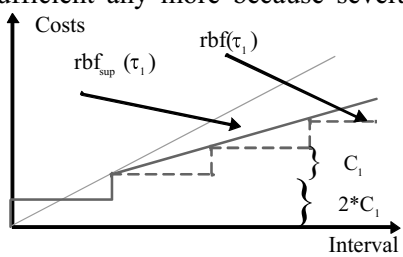


Figure 2. Approximated rbf_{sup} of τ_1

4. EFFICIENT DYNAMIC TEST

To improve the performance of the exact test a combination of the exact test and an approximation is proposed. The idea is to use the approximation for skipping as many test points as possible.

```
function ApproxSchedFunc(τ,Γh,I)
cumureq = Γh.cumu(I,τ)
if (cumureq ≤ I) ⇒ return true
∀τi ∈ Γh : testlist.add(τi, lastJobBefore(I,τi))
approxList() = {}
Uapprox =0
Itest= I
approxReq = cbf_sup'(I, Γ)
while (Itest > I-Tτ)
   τi, Itest = testlist.getLast()
   approxReq = approxReq - Ci - Uapprox *(Itest - Iold)
   if (approxReq ≤ Itest)
        reduceApproximation (approxReq, testList, approxList)
        if (approxReq ≤ Itest)
             ⇒ return true
   if (τi ∈ approxList )
        approxList.remove(τ,)
        Uapprox = Uapprox - Ci/ Ti
   testlist.add(τi, Itest - Ti)
   Iold = Itest
end while
⇒ return false
end function
```

Figure 3. Dynamic Approximation

The test is done using the request bound function. For the feasibility test of one task it is sufficient to show that, for each deadline of the tasks the request bound function for one test interval smaller than the deadline, meets or steps below the intersection. The problem is therefore to find this interval. An overestimating request bound function is not applicable to get an exact test. See Fig 4a for an example. If only the approximated request bound function is regarded the only acceptable test interval would be missed. So in contrast to the approximative test, it is not possible to use rbf_{sup}, the overestimating approximation function. Therefore an underestimating approximation (rbf_{inf}') is used: *A lower bound for the task request bound function considering only the first k jobs exactly*.

$$rbf_{inf}(I, \Gamma_i, k) = \begin{cases} rbf(k \cdot T_i, \Gamma_i) + \frac{C_i}{T_i} \cdot (I - d_{i,k}) - C_i & I > k \cdot T_i \\ rbf(I, \Gamma_i) & I \le k \cdot T_i \end{cases}$$

Consider Fig 4b. The approximation rbf_{inf} is comparable to the approximation rbf_{sup}. In contrary to rbf_{sup} it underestimates the request bound function. In case the test seems to succeed using this approximation it is therefore necessary to reduce the approximation step by step until the exact case is reached. Than the feasibility test with regard to interval *I* succeeds. It is only necessary to calculate the exceeding costs until they meet the additional request. If this is the case, the deadline D_n is met. The algorithms are shown in Fig 3 and Fig 6. The idea of approximation as early as possible can be used for a further improvement of the test. It is necessary to calculate the cumulated costs at least once for each priority level. Evaluating in

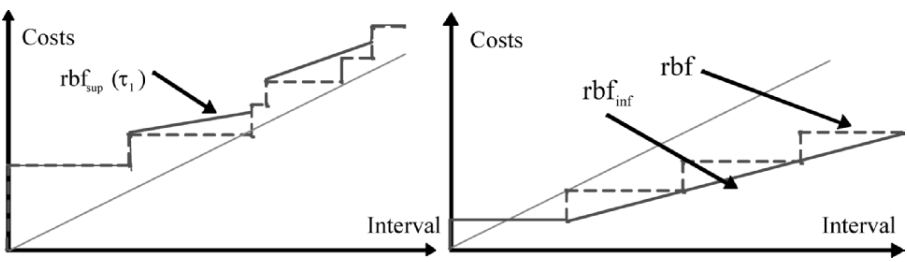


Figure 4. a) Approximated rbf_{sup} of τ_1 b) Approximated rbf_{inf} of τ_1

priority order, an approximative request bound function can be build step-by-step and used as a preliminary check. This allows to quickly check the feasibility for task with low priorities. The idea is to use the upper approximated request bound function for each task with no exact test interval. For any interval I this function can be derived: $rbf_{sup}(I, \tau, 0) = C_\tau + I \cdot C_\tau / T_\tau$

Only one part depends on the length of the interval. It is possible to calculate the sum of the execution times and the sum of the specific utilizations. This calculation can be done step-by-step during the process of testing the different levels of priorities. Evaluating the level in their priority order it is only necessary at each level to add the values for only one task. If the approximated cost exceeds the deadline, the approximated request bound functions are replaced with the exact values for each task step-by-step until either the costs don't exceed the interval any more or all approximations are revised.

```
function reduceApproximation(approxReq, testList, approxList)
  while (approxReq ≤ I_test )
    if (approxList = {})
      return
    τ_i= approxList.getNext()
    approxReq = approxReq + (⌈I_test / T_τ⌉ − I_test / T_τ) C_τ
    U_approx = U_approx - C_i/ T_i
  end while
⇒ return
end function
```

Figure 6. ReduceApproximation

5. EXPERIMENTS

We have tested the proposed algorithms using a large number of randomly generated task graphs. We measured the run-time by counting the iterations needed by the algorithm to find a decision. For the new test we counted every iteration of the main loop, every reverse of an approximation and every iteration needed to get the initial values. The effort is compared to the effort of previous algorithms using the same task sets. For comparison we used an efficient implementation of the worst-case response time analysis introduced by Sjödin and Hansson and of the algorithm for the analysis of recurring real-time task sets proposed by Baruah. We did not consider arbitrary tasks or more advanced task models here. Fig 7a shows an experiment, which uses 50,000 randomly, generated task sets with a utilization between 50% and 99%.

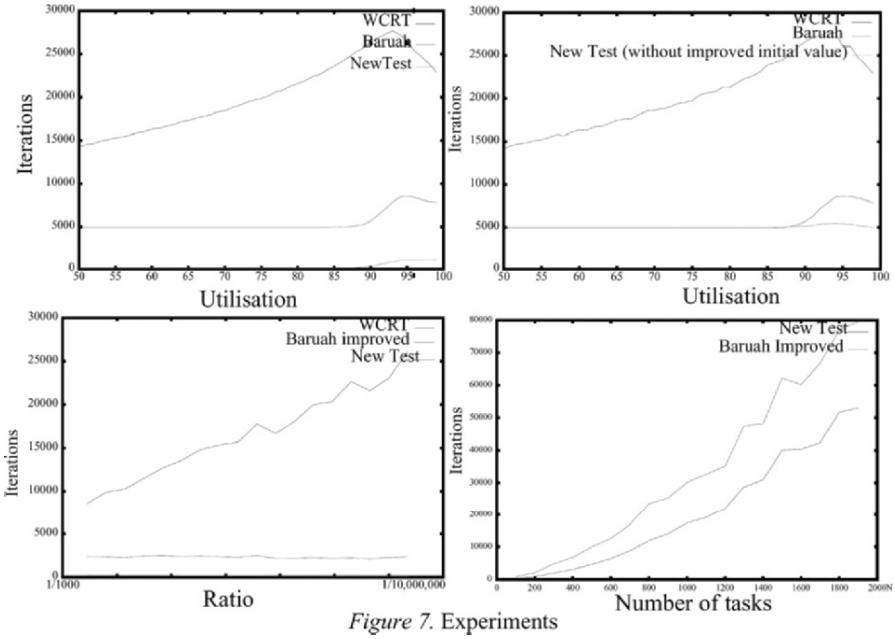


Figure 7. Experiments

The new algorithm shows an improved performance. It outperforms the algorithm by Baruah and only needs between 200 and 1,042 iterations in the average for task sets with different utilizations instead of 4,950 to 8,574. The effort for the worst-case response time analysis even with a very efficient implementation needs between 14,000 and 28,000 iterations and therefore much more than the other algorithms. In cases that the response time is notneeded the new test delivers the result with less effort. To investigate how much of the improvement results only out of the new calculation of the initial value, Fig 7b shows an experiment (using 10,000 task sets) in which the new test does not use the improved approximative calculation of the initial values (Section 4). Despite that both algorithms need a comparable effort for task sets with low utilization, the algorithm by Baruah needs up to 8,500 iterations whereas the new algorithm only needs up to 5,500 iterations. 5,000 iterations are needed alone for calculating initial values. To compare only the impact by the main algorithm an improved version of the test by Baruah was build for the following experiments. It uses also the improved calculation for the initial values. Fig 7a shows an experiment using different ratios between the smallest and the largest tasks in the task set. It is obvious that the effort for the worst-case response time analysis depends on the ratio between the largest and the smallest tasks in the task set. But the effort of the new test and the test by Baruah does not depend on the ratio. Indeed it declines a bit with a rising ratio due to a slightly higher acceptance rate of task sets with a high ratio. In Fig 7b the dependency between the effort and the number of tasks in a task set is shown. To investigate the improvement

due to the dynamic approximation both algorithms use the improved cumulated cost calculation. The results show that both algorithms depend on the number of tasks in a comparable way. Despite this the new algorithm seems to need less effort than the algorithm by Baruah in its improved implementation.

6. CONCLUSION

In this paper we proposed a new fast sufficient and necessary test for the feasibility analysis of preemptive static priority scheduling. Therefore a new approximation for static priority scheduling was also developed. The purpose of the approximation is to limit the run-time of the test. The new exact test, wich also uses the idea of approximation, leads to a lower run-time for feasibility tests, which also seems to be independent of the ratio between the largest and the smallest task.

REFERENCES

K. Albers, F. Slomka. *An Event Stream Driven Approximation for the Analysis of Real-Time Systems.* Proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, 2004

K. Albers, F. Slomka. *Efficient Feasibility Analysis for Real-Time Systems with EDF Scheduling*, Proceedings of the Design Automation and Test in Europe Conference 2005 (DATE‘05), Munich, 2005

N.C. Audsley, A. Burns, M.F. Richardson, A.J. Wellings. *Hard real-time scheduling: The deadline monotonic approach*. Proceedings of the 8th Workshop on Real-Time Operating Systems and Software, 1993

S. Baruah *Dynamic- and Static-priority Scheduling of Recurring Real-Time Tasks*. Real-Time Systems. 24, 2003.

E. Bini, G. Buttazzo, G. Buttazzo. *The Hyperbolic Bound for Rate Monotonic Schedulability.* Proceedings of the Euromicro Conference on Real-Time Systems, 2001.

N. Fisher, S. Baruah. *A polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines*, Proceedings of the 17th Euromicro Conference on Real-Time Systems, Palma de Mallorca, 2005

N. Fisher, S. Baruah. *A Polynomial-Time Approximation Scheme for Feasibility Analysis in Static-Priority Systems with Bounded Relative Deadlines*, Proceedings of the 13th International Conference on Real-Time Systems, Paris, 2005

C. Liu, J. Layland. *Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments*, Journal of the ACM, 20(1), 1973

J.P. Lehoczky, L. Sha, J.K. Stronsnider, H. Tokuda. *Fixed Priority Scheduling Theory for Hard Real-Time Systems.* Foundation of Real-Time Computing: Scheduling and Resource Management, 1991

Y. Manabe, S. Aoyagi. *A Feasibility Decision Algorithm for Rate Monotonic and Deadline Monotonic Scheduling*, Real-Time Systems, 14, 1998

M. Sjödin, H. Hansson. *Improved Response-Time Analysis Calculations*, Proceedings of the RTSS, Madrid, Spain, 1998

APPROACH FOR FORMAL VERIFICATION OF A BIT-SERIAL PIPELINED ARCHITECTURE

Henning Zabel, Achim Rettberg and Alexander Krupp
University of Paderborn/C-LAB, Germany
{henning.zabel, achim.rettberg, alexander.krupp}@c-lab.de

Abstract: This paper presents a formal verification for a bit-serial hardware architecture. The developed architecture bases on the combination of different design paradigms and requires sophisticated design optimizations. The recently patented synchronous bit-serial pipelined architecture, which we investigate in this paper, is comprised of synchronous and systematic bit-serial processing operators without a central controlling instance. We add timing constraints at the boundaris of the architectures operators. To prove the validity of data synchronization we apply formal verification of the constraints on a cycle accurate representation of the implementation. The results of the formal verification is back annotated to the high-level model.

1. INTRODUCTION

Bit serial architectures have the advantage of a low number of input and output lines leading to a low number of required pins. In synchronous design, the performance of those architectures is affected by the long lines, which are used to control the operators and the potential gated clocks. The long control lines can be avoided by a local distribution of the control circuitry at the operator level. Nowadays, the wire delay in chip design is close to the gate delay. In asynchronous design an interlocked mechanism is often used to activate the computation on the desired operator and stall the part of the circuitry, which should be inactive. Reactivation is triggered by a short signal embedded in the datastream. While the design of a fully interlocked asynchronous architecture is well understood, realizing a fully synchronous pipeline architecture still remains a difficult task. By folding the one-hot implementation of the central control engine into the data path, and the use of a shift register, we realized a synchronous fully self-timed bit-serial and fully interlocked pipeline archi-

Zabel, H., Rettberg, A., Krupp, A, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 47–56.

tecture called MACT[1]. Synchronizers of MACT ensure the synchronization of data. Furthermore, routers allow the selection of different paths.

The implementation of the synchronizing elements is an error-prone task. Therefore, we add CTL properties to each MACT operator and apply formal verification to ensure their correctness. We use CTL to describe timing constraints for the I/O of operators. For synthesis, we can fall back on an already realized high level synthesis for MACT [4, 7], which automatically generates MACT implementations in VHDL out of data flow graphs. Additionally, we generate a cycle accurate representation for the model checker from the data flow graph. We investigated how to use the results from the formal verification within the high-level synthesis. Therefore, we identify operators which potentially belongs to a deadlock by tracing violated CTL properties via outputs towards inputs of the MACT model.

The paper is organized as follows. In section 2 the related work is presented followed by the description of the MACT architecture, see section 3. Section 4 presents the formal verification tool RAVEN we used for our approach. The high-level synthesis it self is presented in section 5, followed by a description of the RAVEN specification model, see 6. The verification of the MACT properties are described in 7. Section 8 sums up with a conclusion and gives an outlook.

2. RELATED WORK

Symbolic model checking is well established in the formal verification of small systems, in particular in the field of electronic system design [3]. Compared to classical simulation, the most notable benefit of model checking is the completely automated ellaboration of the complete state space of a system.

The primary input for model checking is a system model given by a set of finite state machines. As a second input, model checking needs a formal specification of required properties. The requirements are typically given as formulae based on temporal logics. Once having specified a property as a temporal logic formula f_i, a model checker can directly answer the question „Does a given model satisfy property f_i?" by either true or false. On request, a model checker typically generates counter examples when the model does not satisfy a specified property. A counter example demonstrates an execution sequence that leads to a situation that falsifies the property, which is very helpful for detailed error analysis.

Recently, Hardware Verification Languages like PSL, SystemVerilog, and *e* have been standardized, which offer assertion flavours for property definition for an underlying Hardware Design Language[6, 5]. Several verification en-

[1]MACT = Mauro, Achim, Christophe and Tom (Inventors of the architecture)

vironments exist for simulation, and for formal property checking from, e.g., Synopsys, Mentor, Cadence, AxiomDA. The current methodology enables the creation and reuse of so-called Verification-IP.[2, 1] Verification-IP represents a library of assertions for system building blocks, basically. The focus of standardized temporal assertions is on linear temporal logic, but branching temporal logic is supported as well, e.g., with PSL OBE.

In our work we automatically generate code from our model for formal verification by the model checker RAVEN. Similar to the Verification-IP-approach, we apply generalized CTL assertions with our intermediate format and use them as formulae for the model checker.

3. MACT ARCHITECTURE

MACT is an architecture, that breaks with classical design paradigms. Its development came in combination with a design paradigm shift to adapt to market requirements. The architecture is based on small and distributed local control units instead of a global control instance. The communication of the components is realized by a special handshake mechanism driven by the local control units. MACT is a synchronous, de-centralized and self-controlling architecture. Data and control information are combined into one packet and are shifted through a network of operators using one single wire only (refer to Figure 1).

The controlling operates locally only based on arriving data. Thus, there exist no long control wires, which would limit the operating speed due to possible wire delays. This is similar to several approaches of asynchronous architectures and enables high operation frequency. Yet, the architecture operates synchronous, thus enabling accurate estimation of latency, etc. a priori.

MACT is a bit-serial architecture. Bit-serial operators are more area efficient than their parallel counterparts. The drawback of bit-serial processing is the increase in latency. Therefore, MACT uses pipelining, i. e., there exist no buffers, operators are placed following each other immediately.

Implementations of MACT are based on data flow graphs. The nodes of these graphs are directly connected, similar to a shift register. Synchronization of different path lengths at nodes with multiple input ports is resolved by a stall mechanism, i. e., the shorter paths, whose data arrives earlier, will be stalled until all data is available (refer to Figure 1). The necessary stall signals run in opposite to the processing direction and are locally limited, in order to avoid a complete stall in the preceding pipeline. The limitation is realized by a so called *block stall* signal, which is locally tapped in a well defined distance.

We consider the flow of data through the operator network as processing in waves, i. e., valid data alternates with gaps. Due to a sophisticated interlock mechanism adapted from asynchronous design, the gap will not fall below

an individual lower bound. Thus, the MACT implements a fully interlocked pipeline. The corresponding signal is the so called *free previous section* signal, which is generated by small logic in each synchronizing control block (see Figure 1). These control blocks are found at each multiple input operator and synchronize packets arriving at different instances of time. The architecture is described in more detail in [8, 9, 7].

4. RAVEN

For formal verification of a MACT model, we apply the RAVEN model checker [10] that supports verification of real-time systems. In the following, we give a brief introduction to RAVEN.

In RAVEN, a model is given by a set of time-annotated extended state machines called I/O-interval structures. I/O-interval structures are based on Kripke structures with [min,max]-time intervals and additional input conditions at their state transitions. A more detailed and formal specification of I/O-interval structures is presented in [11].

For property specifications on a given set of I/O-interval structures, the Clocked Computation Tree Logic (CCTL) is applied [12]. CCTL formulae are composed from atomic propositions in combination with boolean connectives, and time-annotated temporal operators. Timing interval specification is a significant benefit of CCTL when being compared to the classical Computation Tree Logic (CTL) [3].

In the context of RAVEN, I/O-interval structures and a set of CCTL formulae are specified by means of the textual RAVEN Input Language (RIL). A RIL

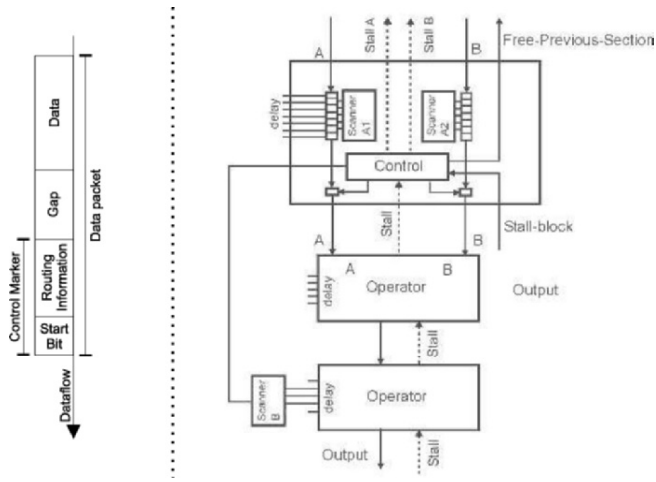


Figure 1. Example data packet (left) and Synchronisation (right)

specification contains (a) a set of global definitions, e.g., fixed time bounds or frequently used formulae, (b) the definition of parallel running modules, i.e., a textual specification of I/O-interval structures, communicating via signals and (c) a set of CCTL formulae, representing required properties of the model.

5. THE MACT HIGH LEVEL SYNTHESIS (MHLS)

MACT models are created upon a predefined set of MACT low level modules. This are operators, delays and synchronisation elements. A MACT model is designed with the graphical MHLS editor. The developer defines the input/output lines and the bit-serial operators. Out of this the MHLS can generate VHDL code for synthesis. For operators the input data packets must arrive synchronous, so that the corresponding data bits are processed at the same clock cylce. For this the MHLS adds automatically synchronisation elements like *Delay* or *DataSync* to the model. Additionally the operators are combined in different sections. These sections prevent the stall conditions from beeing propagated through the complete model (see section 3). Adding this synchronisation elements is a hard task. Wrong placed delays or programming mistakes escpecially in wireing the synchronisation lines can hold off the model from working correctly. In our approach we use model checking techniques to verify that the model conforms to the specification and most important that we don't introduce deadlocks with the stalling mechanism.

To verify this we use the model checker RAVEN. For this we translated the low level MACT modules to a cycle accurate representation in RIL. Withit our code generator composes a complete RIL model description upon the compositional description of the used low level MACT modules. The code generator depends on a XML description. Each module has its own XML file, that contains module descriptions i.e., the names of input and output signals, and a description of the composition of several low level modules or for low level modules the name for the RIL file. This description is very similar to a component description in VHDL and can be generated by the MHLS editor directly. The XML files also contains CCTL formulae with properties over local input and outputs signals, which are automatically added to the RIL file for further processing.

6. SPECIFICATION OF MACT MODULES IN RIL

The lower level MACT modules like *Add*, *Sync* and *Delay* are originally given as a VHDL file. There exist no more than 20 modules and by reason that MACT is a bit-serial architecture these VHDL files are very small too. The effort to tanslate these modules to RIL is manageable, so we have done it by hand.

When mapping these modules to RIL, we have to find a representation for signals, processes and the hierarchie in components. Signals can be devided in clock dependent and clock independent signals.

We translated each VHDL process as RIL module. Figure 2 shows an example of VHDL code and its corresponding RIL code. By instanziating the

```
stall_out <= stall_in;

and : process (CLK,RESET)
begin
  if RESET = '1' then
    summ  <= '0';
  elsif CLK'event and CLK = '1' then
    summ  <= line_0_in & line_1_in;
  elsif
    summ <= summ;
  endif
end
```

```
MODULE and:
  SIGNAL
    rv  :BOOL
  INPUT
    line_0_in := ?
    line_1_in := ?
    stall_in  := ?
    reset     := ?
  DEFINE
    // -- outputs --
    line_out  := summ
    stall_out := stall_in

    // -- boolean term for rv --
    rv_cond := !reset /\
           (line_0_in /\ line_1_in)
  TRANS
    |- TRUE -- :1 --> rv := rv_cond
END
```

Figure 2. VHDL module (left) and its representation in RIL (right)

processes the hierarchie can be flattenend. We extended the signal names with the process names to keep unique identifiers for the signals.

In RIL all signals change their value only within a transition step. In our specification one transition step in RIL corresponds to one clock cycle in the VHDL design. Thus clocked signals can be represented as RIL signals in their corresponding RIL module. Clock independent signals are defined as boolean terms over other signals. It is always true, that their value is present at the next rising edge of the clock. Otherwise the design is overclocked in relation to the longest path. If all the clock independent signals belongs to clocked signals as source, then these signals can be represented as a DEFINE in RIL. Because in this case they are only placeholders for their source signals with an additional boolean term.

This representation is only true, if there are no logic cycles within clock independent signals. But only the *Sync* module has logic cycles without a clock to model a RS-flipflop. All other clock independent signals are defined over clocked ones, but generally not within the same module.

7. VERIFICATION OF MACT PROPERTIES

MACT models are triggered by a leading 1 within the datastream. Usually, MACT is used for pipelined data processing, so every valid input packet to a MACT model should lead to a valid output packet. If the model gets no input, then it shouldn't produce output. For synchronisation, data packets can

be stalled within a MACT model. To avoid deadlocks it is important to prove that these stalls always disappear.

7.1 CTL formulae

Most of the MACT lowlevel modules only pass through the stall informations. Stalls itself are only generated in the *DataSync* and *Sync* modules. The following CCTL properties have to be true for a deadlock free operation of a specific model. In general all MACT modules must hold these properties, but specially the synchonisation modules.

1 We always get data, eventually: $AG(!line\_in \rightarrow AF(line\_in))$

2 We never stall our inputs forever: $AG(stall\_out \rightarrow AF(line\_in \rightarrow AF(!stall\_out)))$

3 We never get stalled forever: $AG(stall\_in \rightarrow AF(!stall\_in))$

4 We always output data, eventually: $AG(!line\_out \rightarrow AF(line\_out))$

There are some relations between these formulae: At first Property 1 and 4 are true, if valid data arrives or leaves. But this is also true if the module is stalled, because the formulae only grants that data arrives (leading 1), not disappears. At second, Property 2 only makes sense, if property 1 is true. This is because the second implication is also true even if no data arrive and thus the stall will never disappear.

7.2 Detecting stalling modules

The codegenerator adds synchronisation elements to the implementation of the MHLS model (see figure 3). All these added elements can be associated with one module in the MHLS model. Because of this, we can detect stalling elements in the MHLS model by detected stalling modules in the implementation. Although stalling elements are introduced during codegeneration, this is a usefull benefit to localise the deadlock.

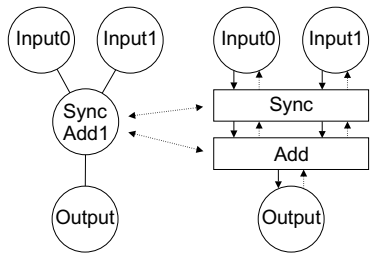


Figure 3. MHLS Model and implementation model

At first the above mentiond formulae should be checked for all inputs and outputs. If they are all true, then we know that data passes the MACT model and we can stop here. If we encounter a problem, the model checker can produce a counter example. For big models it is very hard to localise the source of a deadlock simply by looking at the waveforms (counter example).

An easier way is to check the above formulas step by step on every module from the outputs up to the inputs by repeating the following steps. They will detect and mark all modules in the MHLS model, that are involved in a deadlock.

1 if the module is already involved in a deadlock (marked) then return

2 check all formulas of the local module

3 if the module itself is stalled by its successor (formula 3) then this module can be marked and the algorithm returns

4 if one of the inputs gets stalled (formula 2) or sends no data (formula 1), then mark them and proceed recursive with it.

This algorithm can be realised by the use of RAVENs interactive mode. It allows to start RAVEN with a complete model and different CCTL formulae. The verification of a specific formula can be triggered via a command line interface.

7.3 Example

To prove our method, we use a MACT model that adds four numbers like shown in figure 4. The inputs get valid MACT packets in well defined intervals, expcept input three: The decision wether a packet is send or not is nondeterministic, that means there exists execution pathes where no packet will be send.

We generated RIL out of the model and invoked RAVEN. RAVEN needs 40ms to verify this model on an Pentium4 with 3GHz and creates a transition table with 1841 BDD Nodes. The result of the formulae is shown in figure 5.

All checks for receiving and sending data are false for the successors of input three up to the output. If we apply our algorithm from 7.2, the grayed operators in figure 5 are marked. These are exactly the modules involved in the deadlock. Of course *add0* and *add1* stalls their other inputs to wait for data on the other line, but anyway formulae „don't stall input forever" is true. This is because the second implication in the formulae, which requires a valid input. In our case this doesn't occur, so the implication is always true.

We also tested this apporach on a much bigger model of a discrete cosinus transformation (DCT), but RAVEN breaks the limit of 4GB memory, when it generates it's internal data structure. Thus, for larger models an abstraction is

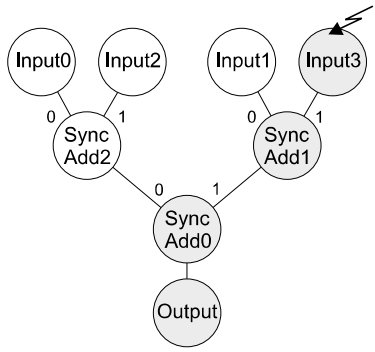


Figure 4. Example Model

Formulae	add0		add1		add2	
	line 0	line 1	line 0	line 1	line 0	line 1
always get data	true	false	true	false	true	true
never stall input forever	true	true	true	true	false	false
never get stalled forever	true	true	true	true	false	false
always output	false	false	false	false	true	true

Figure 5. Model checking results

needed. For example the operators can be abstracted as simple delays, because the operation itself has no direct influence on the stalling mechanism.

8. CONCLUSION AND OUTLOOK

We presented an approach to generate a cycle accurate representation of VHDL implementation of the bit-serial architetcture MACT in RIL. It is composed of small sub-modules and we automatically generate CTL formulae for them to check local data flow properties. Based on this formulae we presented a depth search algorithm to detect stalling modules.

We showed that this approach is applicable for small models and it finds the stalling path. For larger models an abstracted is needed, that only represents the cycle accurate stalling mechanism.

We plan to integrate this algorithm in our MHLS editor. With it we can visualise the modules that are involved in the deadlock. This should help the developer for the MHLS code generator to detect programming errors. It also helps the developer of a MACT model to integrate sufficient delays to its model.

The automated generation of CCTL formulae for each module can also be applied to verify timing constraints of the MACT model too. Exspecially timing that belongs to the length of a data packet is important and could be in-

tegrated during the high level syhntheses. RAVEN has special extensions for such timing constraints.

REFERENCES

[1] M. Bartley, D. Galpin, and T. Blackmore. A comparison of three verification techniques: Directed testing, pseudo-random testing and property checking. In *Proceedings of DAC 2002*, New Orleans, June 2002. ACM.

[2] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer, 2006.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[4] Florian Dittmann, Achim Rettberg, Thomas Lehmann, and Mauro C. Zanella. Invariants for distributed local control elements of a new synchronous bit-serial architecture. In *Second IEEE International Workshop on Electronic Desing, Test and Applications (DELTA 2004)*, pages 245–250, Perth, Western Australia, 28 - 30 January 2004.

[5] IEEE. *IEEE Std.1800-2005 - Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language*, November 2005.

[6] IEEE. *IEEE Std.1850-2005 - IEEE Standard for Property Specification Language (PSL)*, September 2005.

[7] Achim Rettberg, Florian Dittmann, Mauro C. Zanella, and Thomas Lehmann. Towards a high-level synthesis of reconfigurable bit-serial architectures. In *Proceedings of the 16th Symposium on Integrated Circuits and System Design (SBCCI)*, Sao Paulo, Brazil, 8 - 11 September 2003.

[8] Achim Rettberg, Thomas Lehmann, Mauro C. Zanella, and Christophe Bobda. Selbstsynchronisierende rekonfigurierbare bit-serielle pipelinearchitektur. Deutsches Patent- und Markenamt, December 2004. Patent-No. 10308510.

[9] Achim Rettberg, Mauro C. Zanella, Christophe Bobda, and Thomas Lehmann. A fully self-timed bit-serial pipeline architecture for embedded systems. In *Proceedings of the Design Automation and Test Conference (DATE)*, Messe Munich, Munich, Germany, 3 - 7 March 2003.

[10] J. Ruf. "RAVEN: Real-Time Analyzing and Verification Environment". *Journal on Universal Computer Science (J.UCS), Springer, Heidelberg*, February 2001.

[11] J. Ruf and T. Kropf. "Modeling and Checking Networks of Communicating Real-Time Processes". In *Int. Conf. on Correct Hardware Design and Verification Methods*, Bad Herrenalb, Germany, 1999. Springer-Verlag.

[12] Jurgen Ruf and Thomas Kropf. "Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals". In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, Montreal, Canada, October 1997.

AUTOMOTIVE SYSTEM OPTIMIZATION USING SENSITIVITY ANALYSIS

Razvan Racu, Arne Hamann, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
D-38106 Braunschweig, Germany

Abstract: There have been major advances in formal methods and related tools in embedded system design in recent years that support analysis and optimization of heterogeneous automotive architectures. We demonstrate the application of the SymTA/S methodology to an automotive example where we analyze the sensitivity of different system properties and determine optimal system configurations. The use of this methodology in the early phases of system design conduct to results that cannot be obtained using simulation or prototyping.

1. INTRODUCTION

With increasing automotive system complexity, formal methods become more important as a complement to simulation and prototyping. Formal methods have already been used in early architecture design, e.g. to optimize bus architectures as such models can be applied before executable models are available. Recent advances in real-time system analysis have extended the scope of formal models to heterogeneous networks with different protocols and gateways, and with electronic control units (ECUs) running different scheduling algorithms. These formal models can be used to optimize those systems and analyze the sensitivity to later changes in the design process or to run time events such as retransmission due to errors. In this paper, we give a brief introduction to the underlying model and algorithms of the tool, SymTA/S, and give a small example for its application to system integration.

The remainder of this paper is structured as follows. First, we will give a short overview of existing formal techniques for system level performance analysis (Section 2). Afterwards, we briefly present the SymTA/S compositional analysis methodology based on event model interfacing and propagation (Section 3).

In Section 4 we present the main scheduling concepts of the ERCOSEK operating system and the CAN bus protocol.

Racu, P., Hamann, A., Ernst, R, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 57–70.

Finally, we introduce a small example of an automotive system consisting of two independent subsystems (Section 5) and demonstrate the application of the SymTA/S exploration and sensitivity analysis frameworks to system integration (Sections 6 to 9).

2. FORMAL TECHNIQUES IN SYSTEM PERFORMANCE ANALYSIS

In this section, we briefly review existing analysis approaches from real-time research for formal system performance analysis of heterogeneous distributed systems and MpSoC.

The holistic analysis approach developed by Tindell [12] systematically extended the classical local analysis techniques, considering the scheduling influences along functional paths in the system. He proposed a performance verification model for distributed real-time systems with preemptive task sets communicating via message passing and shared data areas. Eles *et al.* [7] extended this approach for systems consisting of fixed-priority scheduled CPUs connected via a TDMA scheduled bus. Later on, Palencia *et al.* [6] extended the analysis for tasks with precedence relations and activation offsets.

Gresser [4] and Thiele [11] established a different view on scheduling analysis. The individual components or subsystems are seen as entities which interact, or communicate, via event streams. Mathematically speaking, the stream representations are used to capture the dependencies between the equations (or equations sets) that describe the individual components timing. The difference to the holistic approach (that also captures the timing using system-level equations) is that the compositional models are well-structured with respect to the architecture. This is considered a key benefit, since the structuring significantly helps designers to understand the complex dependencies in the system, and it enables a surprisingly simple solution. In the "compositional" approach, an output event stream of one component turns into an input event stream of a connected component. Schedulability analysis, then, can be seen as a flow-analysis problem for event streams that, in principle, can be solved iteratively using event stream propagation.

3. THE SYMTA/S APPROACH

SymTA/S [10] is a formal system-level performance and timing analysis methodology for heterogeneous SoCs and distributed systems. The key novelty of the SymTA/S approach is that it uses intuitive event models from real-time system research to describe the timing behavior the activating events, rather than introducing new, complex stream representations. SymTA/S uses a six-class *standard event models* and a mathematical formalism that easily allow the interfacing and transformations between models [9].

Periodic events or event streams with jitter and bursts are examples of standard models that can be found in literature. The SymTA/S technology extracts

the stream parameters from a given schedule and automatically adapts the event stream to the specific needs within these standard models, so that designers can safely apply existing subsystem techniques of choice without compromising global analysis.

The compositional performance analysis methodology defined in SymTA/S [9] alternates local scheduling analysis and event model propagation during system-level analysis. The system analysis is performed iteratively in several global analysis steps. A global analysis step consists of two phases. In the first phase local scheduling analysis is performed for each resource and output event models are calculated. In the second phase, all output event models are propagated. Then, it is checked if the first phase has to be repeated because some activating event models are no longer up-to-date, meaning that a newly propagated output event model is different from the output event models that was propagated in the previous global analysis step. Analysis completes if either all event models did not change during last propagation phase, or an abort condition, e. g. the violation of a timing constraint, has been reached.

The propagation phase requires the modeling of possible timing of the output events of a component. The standard event models allow to specify simple rules to obtain output event models from the parameters of the input event models, considering the scheduling influences.

4. AUTOMOTIVE EMBEDDED TECHNOLOGY

In the following sections we give a brief overview about the most important scheduling concepts of ERCOSEK operating system and CAN bus protocol. Both arbitration techniques are well established in the automotive industry and deployed in various car platforms and product lines.

4.1 ERCOSEK

The ERCOSEK [3] operating system builds upon the core ideas of static-priority preemptive scheduling. However, this underlying scheduling policy is extended by a variety of additional concepts.

ERCOSEK distinguishes *hardware tasks (interrupts)*, *preemptive software tasks* and *cooperative software tasks*. Software tasks are comprised of processes that are sequentially executed. In contrast to preemptive software tasks, cooperative software tasks are preemptive only at process boundaries. This reduces the context switch overhead but results in additional blocking for the higher priority cooperative tasks.

Certain scheduling-related OS routines can request a considerable amount of execution time at various priority levels. For instance, the *activate task* and *terminate task* routines are called by the OS before and after task execution, respectively. Both OS routines are executed with the so-called *kernel priority*, which is higher than the priority of all software tasks.

Furthermore, task activation can be initiated in a variety of ways. *Time tables* allow the specification of periodic tasks with phase offsets (startup delays) between them. *Alarms* use more dynamic time-out mechanisms and can be issued and disabled at any point in time. Finally, tasks can be activated from software tasks and interrupts (hardware tasks) dynamically and bursty.

4.2 CAN (Controller Area Network)

The CAN bus protocol [2] is also based on static priorities but the message transmission is non-preemptive, as typical for serial line protocols.

Compared to ERCOSEK's characteristics, the actual CAN protocol is relatively simply. However, due to cost reasons, CAN interfaces are typically realized with a very limited number of sender buffers, so-called *message objects*. A message, once written into such a buffer, can not be "overtaken" by another higher-priority message that is generated later. This behavior can turn the static-priority scheme into a complex queuing scheme when it comes to scheduling analysis.

Furthermore, CAN messages inherit the time-table-like behavior of the tasks that generate the messages. In combination with dynamic phase shifts between request and acknowledge frames in an end-to-end path this leads to complex best-case and worst-case scheduling scenarios.

Finally, transmission errors can enforce retransmissions that increase the overall load and message latency.

5. AUTOMOTIVE EXAMPLE SYSTEM

Figure 1 shows a SymTA/S model of two electronic subsystems. The two subsystems are functionally independent, and are designed separately by two different electronic suppliers.

The automotive OEM would like to integrate both subsystems in a vehicle.

Note that $ECU1$ is arbitrated by the ERCOSEK operating system, whereas all other ECUs are arbitrated according to the static priority preemptive policy (SPP). The buses used to exchange messages between the ECUs in both subsystems are running the CAN protocol.

The core execution times of the software-functions running on the ECUs are given in Table 1. The communication delays of the messages exchanged over the CAN buses, assuming no concurrent communication requests, are given in Table 2. Tables 1 and 2 additionally contain the priorities of each software-function as well as the IDs of the CAN messages (representing also the priorities). Except for the ERCOSEK scheduled software-functions lower values correspond to higher priorities.

The gray, rounded boxes model the activation of the software-functions (in this case, timers). The activation periods are given in table 3.

Note that both subsystems need to satisfy certain timing constraints in order to function correctly:

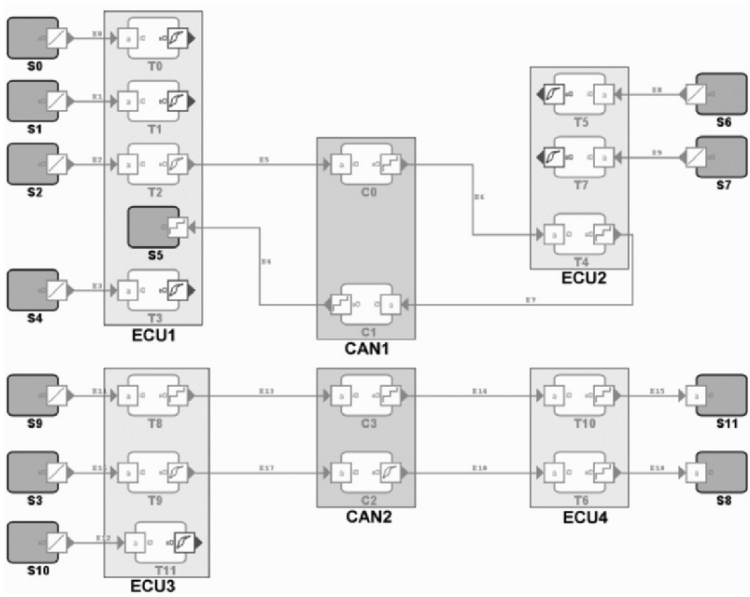


Figure 1. Automotive system example: independent subsystems

Table 1. Computational Tasks

Task	*Execution time*	*Priority*
	ECU1 (ERCOSEK)	
T0	[0.2,0.3]	10 (preemptive)
T1	[0.2,0.3]	5 (preemptive)
T2	[0.1,0.2]	2 (cooperative)
T3	[1.2,2.3]	1 (cooperative)
	ECU2 (SPP)	
T4	[0.7,0.8]	2
T5	[0.1,0.2]	3
T7	[0.1,0.6]	1
	ECU3 (SPP)	
T8	[0.3,0.4]	2
T9	[1,2]	1
T11	[3,5]	3
	ECU4 (SPP)	
T6	[1.1,1.5]	2
T10	[0.3,0.6]	1

– path $S2 \rightarrow S5$ has an end-to-end deadline equal to 15.
– path $S9 \rightarrow S11$ has an end-to-end deadline equal to 15.
– path $S3 \rightarrow S8$ has an end-to-end deadline equal to 15.

Figure 1 shows both subsystems before integration. Both the upper and the lower subsystem are implemented on 2 ECUs (vertical square boxes on the left

Table 2. Communication Tasks

Channel	*Communication time*	*Priority*
CAN1 (Controller Area Network)		
C0	[1.08,1.32]	1
C1	[0.76,0.92]	3
CAN2 (Controller Area Network)		
C2	[0.6,0.72]	4
C3	[0.68,0.82]	2

Table 3. Input Event Models

Input	*Event Model*	*Parameters*
$S0$	periodic	$\mathcal{P}_1 = 1$
$S1$	periodic	$\mathcal{P}_1 = 2$
$S2$	periodic	$\mathcal{P}_2 = 5$
$S3$	periodic	$\mathcal{P}_3 = 15$
$S4$	periodic	$\mathcal{P}_3 = 20$
$S6$	periodic	$\mathcal{P}_3 = 5$
$S7$	periodic	$\mathcal{P}_3 = 7$
$S9$	periodic	$\mathcal{P}_9 = 2$
$S10$	periodic	$\mathcal{P}_{10} = 10$

and right) that exchange messages over a dedicated CAN bus (vertical square boxes in the middle).

Timing and performance analysis of both subsystems reveals the following worst-case delays for the constrained paths:

– 12.01 time units for the path $S2 \rightarrow S5$

– 5.12 time units for the path $S9 \rightarrow S11$

– 9.92 time units for the path $S3 \rightarrow S8$

If we compare these worst-case delays with the deadlines imposed by system specification, we observe that both suppliers implemented their subsystems so that all timing constraints are satisfied.

6. SENSITIVITY ANALYSIS

The robustness of an architecture to parameter changes is a major concern in the design of embedded real-time systems. Robustness is important in early design stages to identify if and in how far a system can accommodate later changes or updates. Furthermore, system robustness is an important metric in the later design phases during subsystem or third-party component integration. In general, the system robustness is defined by the available headroom or slack corresponding to different properties of the system. These can be task execution demands, channel communication times, the parameters of the acti-

vation models, the speed of the computation resources or the throughput of the communication buses.

Sensitivity analysis allows the system designer to permanently keep track of the system robustness, and thus to quickly asses the impact of changes of individual hardware or software components on system performance. Section 6.1 gives a short overview on the sensitivity analysis framework implemented in SymTA/S. In Section 6.2 we determine the robustness of the independent subsystems presented in Figure 1 with respect to variations of different parameters.

6.1 Sensitivity Analysis Framework

The sensitivity analysis framework implemented in SymTA/S combines a binary search technique and the compositional analysis model presented in Section 3. The binary search technique is known as a simple and fast search algorithm used to determine a specific values within an ordered set of data. Since the variations of specific system parameters like execution demands, activation periods, resource speeds have a monotonic impact on the set of system timing properties, the binary search can quickly determine the values of these parameters that leading to conforming system configurations. A detailed description of the sensitivity analysis framework is presented in [8].

Parameters for sensitivity analysis are the system properties that may vary during design process. Very common are variations of execution times, the parameters of the activation models, like period, jitter and offset, communication volumes, bus and processor speeds. The variations of these parameters affect different system performance metrics, like task response times, end-to-end latencies, output jitters, buffer sizes or deadline miss-ratio in case of soft real-time systems.

6.2 The Robustness of the Independent Subsystems

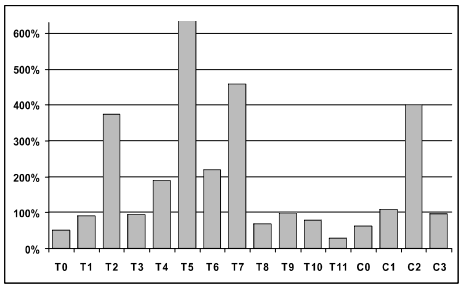
In this section we determine the robustness of the independent subsystems presented in Figure 1. Firstly, we investigate the available slacks of the execution times of the tasks mapped on the ECUs and of the communication channels mapped on the CAN resources. The results are presented in Figure 2(a). We observe that tasks $T2$, $T5$, $T7$ and channel $C2$ have a very large flexibility compared with the other tasks and channels. This is explained by the fact that these tasks belong to functional paths without or with loose timing constraints.

Figure 2(b) shows the flexibility of the operational speeds of computation and communication resources. The minimum speed of these resources is determined on one hand by the utilization factor, and, on the other hand, by the timing constraints defined for the tasks executed on these resources.

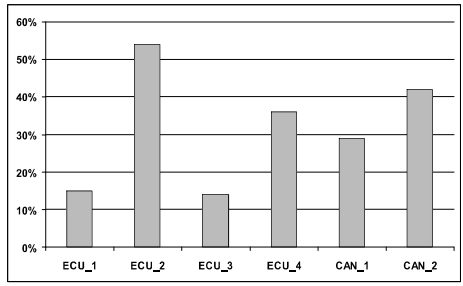
The last investigated set of parameters are the task execution rates defined at system inputs. Figure 2(c) shows the maximum decrease permitted for task

activation periods without violating the set of constraints or the system schedulability.

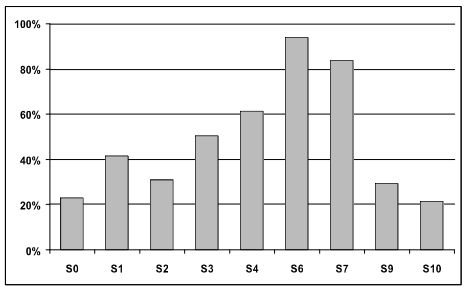
The resulting slacks of investigated system parameters represent an additional argument for the feasible integration of the two independent subsystems. The available resource headroom allows all messages to be transmitted on a single bus without disturbing too much the performance of the other system components.



(a) WCET flexibility



(b) Resource speed flexibility



(c) Activation period flexibility

Figure 2. Flexibility of the initial system configuration

7. INTEGRATING BOTH SUBSYSTEMS

We now integrate both independent subsystems. Figure 3 shows the system after integration. Instead of utilizing a dedicated CAN bus for each of the subsystem, all messages are now transmitted over a single CAN bus. Of course, this leads to additional load and potentially longer blocking of low-priority messages.

Again we verify performance and timing of the system with SymTA/S, and obtain the following worst-case delays for the constrained paths:

– 20.18 time units for the path $S2 \rightarrow S5$

– 7.92 time units for the path $S9 \rightarrow S11$

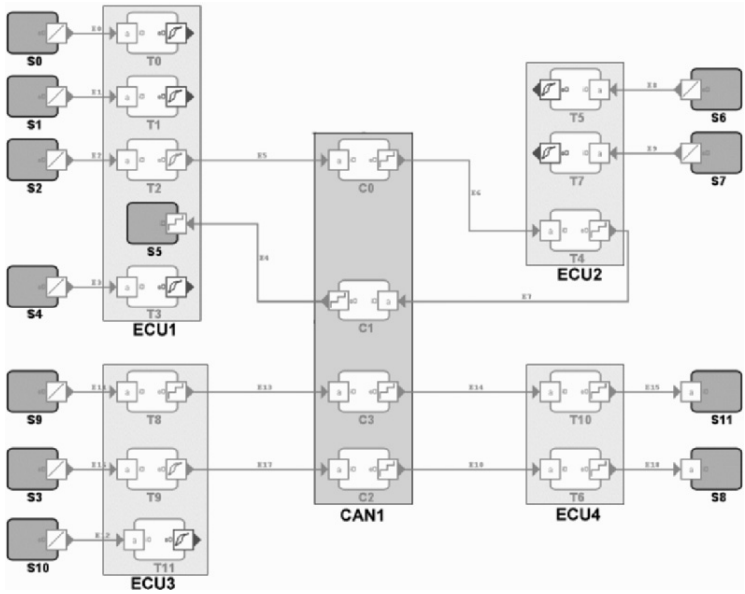


Figure 3. Automotive system example: after system integration

– 33.09 time units for the path $S3 \rightarrow S8$

We observe that the constrained paths $S2 \rightarrow S5$ and $S3 \rightarrow S8$ exceed their deadlines by 34.5% and 120.6%, respectively.

8. SYSTEM OPTIMIZATION

In section 7 we have seen that the integration of two independently working subsystems is usually not possible in a straight-forward manner. In practice, system parameters like CAN message IDs, task priorities, and time slots need to be adapted to successfully integrate several subsystems.

In the following we first introduce the design space exploration framework of SymTA/S, assisting the designer in exploring the configuration space of complex distributed systems and pointing out pareto-optimal system configurations with respect to an arbitrary numbers of optimization criteria, including timing properties, power consumption, buffer sizes, etc. In the second part, we use this framework to explore the integrated automotive example system.

8.1 Design Space Exploration Framework

Figure 4 shows the design space exploration framework [5] of SymTA/S. The *Optimization Controller* is the central element. It is connected to the scheduling analysis of SymTA/S and to an evolutionary multi-objective optimizer. SymTA/S checks the validity of a given system parameter set, that is represented by an individual, in the context of the overall heterogeneous sys-

tem. The evolutionary multi-objective optimizer is responsible for the problem-independent part of the optimization problem, i.e. elimination of individuals and selection of interesting individuals for variation. Currently, we use SPEA2 (Strength Pareto Evolutionary Algorithm 2) [13] for this part, which is coupled via PISA (Platform and Programming Language Independent Interface for Search Algorithms) [1].

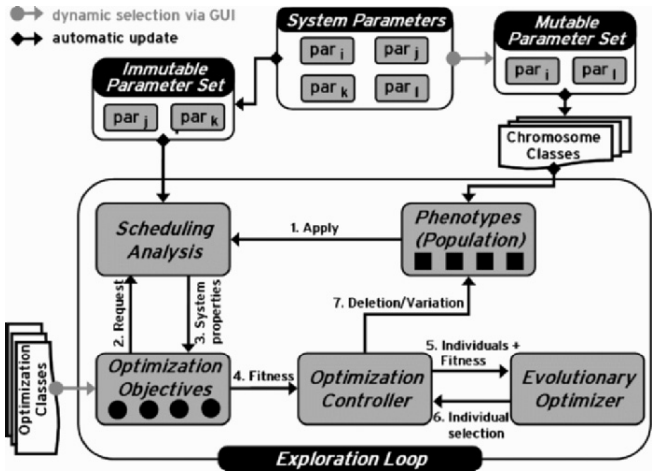


Figure 4. Exploration Framework

Different parameters of a system, such as priorities or time slots, are encoded on separate *chromosomes*. The user selects a set of parameters for optimization. The chromosomes of these parameters form an *individual*, and are included in the evolutionary optimization while all others are fixed and immutable. The variation operators of the evolutionary algorithm are applied chromosome-wise for these individuals.

8.2 Exploring the integrated system

In this section we use the design space exploration framework of SymTA/S to explore the integrated automotive system in Figure 3.

The search space of our exploration consists of the priority assignments on all ECUs as well as the assignment of CAN message IDs on the bus. Since the straight-forward integration of the two subsystems lead to the violation of hard timing constraints, our primary optimization objective is to find a working system configuration. Additionally we are interested in pareto-optimal trade-offs between the three constrained end-to-end paths.

In the real world, no single design team has full control over the entire system. Instead, numerous design teams from different companies contribute along an automotive supply chain. Each team is in control of only part of the system. Therefore, system-level exploration across team-boundaries is a com-

plicated task. The OEM as the bus integrator, for instance, usually controls CAN message IDs, but has very limited insight into the configuration of the ECUs.

Unfortunately, dynamic behavior between several subsystems usually cannot be observed until late in the design process when first ECU prototypes become available. By that time, it is very costly to re-assign system parameters like priorities of software-functions or CAN message IDs.

In order to account for the difficulty and the high cost of system parameter modifications at system integration time, we add the minimization of parameter changes to the optimization objectives considered during design space exploration. More precisely, we are interested in finding working system configurations with as few parameter changes as possible.

Table 4 shows the pareto-optimal system configurations obtained by an exploration considering 1000 system configurations (40 generations with 25 individuals each). Note that this exploration took approximately 70 seconds on a standard PC running at a clock-rate of 2.4 GHz.

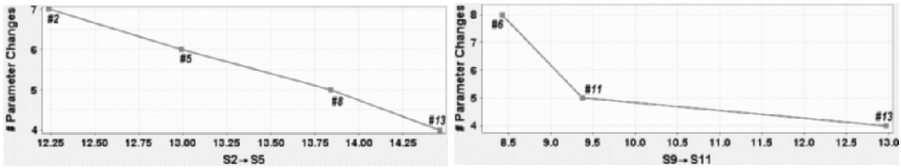
Table 4. Pareto-optimal solutions

#	$S2 \rightarrow S5$	$S9 \rightarrow S11$	$S3 \rightarrow S8$	*# param. changes*
1	12.24	10.12	11.52	9
2	12.24	13.72	10.72	7
3	12.86	10.12	11.52	7
4	12.99	10.12	8.87	8
5	12.99	13.72	8.07	6
6	13.84	8.42	12.97	8
7	13.84	10.12	6.57	10
8	13.84	12.12	14.57	5
9	13.84	13.82	5.77	7
10	14.06	9.37	12.97	7
11	14.46	9.37	12.97	5
12	14.46	11.07	6.57	7
13	14.46	12.97	14.57	4

We observe that we found 13 pareto-optimal working system configurations, each of them representing an optimal trade-off between the constrained timing properties and the number of parameter changes with respect to the initial configuration.

In order to decide which system configuration to adopt, the system integrator needs to interpret the pareto-set. Depending on special requirements to the system she can consider additional information such as system sensitivity to property variations (see Section 9) to make a decision.

Figures 5(a) and 5(b) show the 2-dimensional Pareto-fronts representing the optimal trade-offs between the constrained timing properties and the number of parameter changes necessary to achieve them.



(a) Pareto-Front: #Param changes - $S2 \rightarrow S5$

(b) Pareto-Front: #Param changes - $S9 \rightarrow S11$

Figure 5. Optimization results

If we consider, for instance, the path $S2 \rightarrow S5$, we can see from the pareto-front in figure 5(a), that the minimum number of necessary parameter changes to obtain a working system is 4 (config. $\#13$), corresponding to an delay of 14.46 time units, which is short of the constraint (15 time units).

Increasing the number of allowed parameter changes leads to shorter end-to-end delays. With 5 (config. $\#8$) and 6 (config. $\#5$) parameter changes we can obtain end-to-end delays of 13.84 and 12.99 time units for the path $S2 \rightarrow S5$, respectively. The shortest end-to-end delay for the path $S2 \rightarrow S5$, 12.24 time units, can be achieved with a minimum number of 7 parameter changes.

9. SENSITIVITY ANALYSIS OF THE INTEGRATED SYSTEM

In this section we determine the sensitivity of the pareto-optimal system configurations presented in Section 8.2. Figure 6 shows the flexibility of the task execution times and channels communication times. From the set of pareto-optimal configurations we removed those that dominate the other configurations in at least one computed parameter slack. At a closer look we observe that configurations $\#4$ and $\#7$ determine the maximum available slack for most execution times. In general, comparing the results presented in Figure 6 with the results obtained for the two independent subsystems (Figure 2(a)) we observe that the slacks of the execution times of the system tasks have decreased only by a small amount after system integration. The slack of the communication channels has evidently decreased due to the load increase of $CAN1$.

Figure 7 shows the slack values obtained for the hardware resource speed. Again, we selected only those configurations that have a high overall robustness or those which dominate all other configurations in at least one computed parameter slack. Compared to the results presented in Figure 2(b) the only resource with a noticeable smaller slack is the communication bus, $CAN1$. The reason is again the increase of the overall resource utilization after system integration.

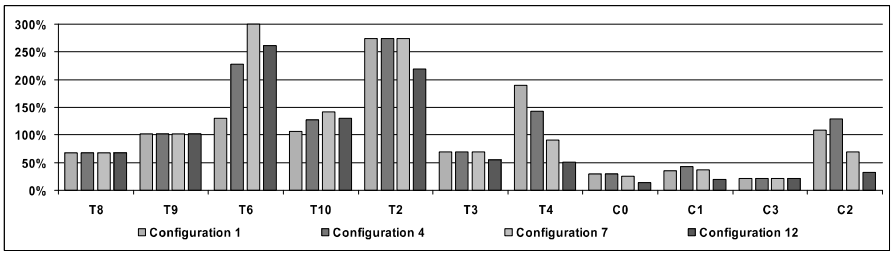


Figure 6. WCET Flexibility

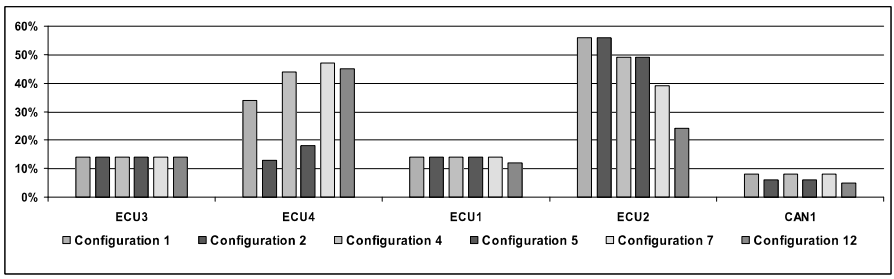


Figure 7. Resource Speed Flexibility

Lastly, we determine the available slack corresponding to the task activation periods. Figure 8 shows the values obtained for some configuration selected from the set of pareto-optimal configurations obtained in Section 8.2. Comparing these results with the results obtained before system integration (Figure 2(c)) we observe that the only periods whose slacks clearly decreased are $S2$, $S3$ and $S9$. Since these periods obviously determine the communication rates of the channels on $CAN1$ and, consequently, automatically the utilization of this bus, and since the overall load on $CAN1$ has increased after subsystem integration, the available headroom of these periods decreased accordingly.

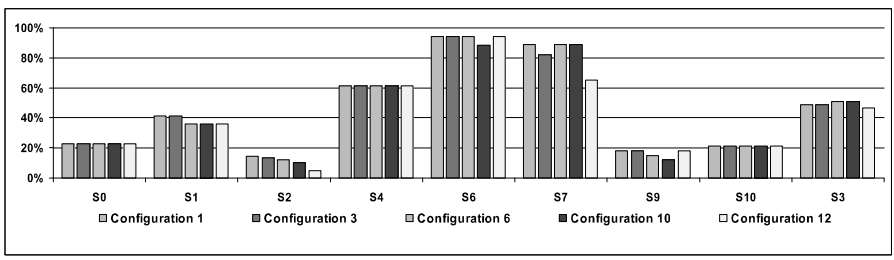


Figure 8. Activation Period Flexibility

10. CONCLUSION

Formal models are an ideal basis to determine system properties that are not amenable to simulation, such as system robustness, and they allow rapid design space exploration. In this paper we showed how sensitivity analyis

can help in quickly dimensioning and optimizing automotive platforms. In the future, formal models based techniques are considered to play a major role in automotive design.

REFERENCES

[1] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA – a platform and programming language independent interface for search algorithms. `http://www.tik.ee.ethz.ch/pisa/`.

[2] CAN in Automation (CiA). *Controller Area Network*. `http://www.can-cia.org`.

[3] ETAS. $ERCOS^{EK}$ *Real-Time Operating System*. `http://www.etas.de`.

[4] K. Gresser. An event model for deadline verification of hard real-time systems. In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 118–123, Oulu, Finland, 1993.

[5] A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design space exploration and system optimization with SymTA/S - Symbolic Timing Analysis for Systems. In *Proc. 25th International Real-Time Systems Symposium (RTSS'04)*, Lisbon, Portugal, December 2004.

[6] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.

[7] Traian Pop, Petru Eles, and Zebo Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, USA, May 2002.

[8] Razvan Racu, Marek Jersak, and Rolf Ernst. Applying sensitivity analysis in real-time distributed systems. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, USA, 2005.

[9] K. Richter. *Compositional Performance Analysis*. PhD thesis, Technical University of Braunschweig, 2004.

[10] SymTA/S - Symbolic Timing Analysis for Systems. `http://www.symta.org`.

[11] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, Geneva, Switzerland, 2000.

[12] K. Tindell and J. Clark. Holistic schedulability analysis for distributed real-time systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[13] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 2001.

TOWARDS A DYNAMICALLY RECONFIGURABLE AUTOMOTIVE CONTROL SYSTEM ARCHITECTURE

Richard Anthony[1], Achim Rettberg[2], Dejiu Chen[3], Isabell Jahnich[2], Gerrit de Boer[4], Cecilia Ekelin[5]
[1]*The University of Greenwich, England*
[2]*University of Paderborn/C-LAB, Germanyy*
[3]*Royal Institute of Technology, Sweden*
[4]*Bosch GmbH, Germany*
[5]*Volvo Technology AB, Sweden*

Abstract: This paper proposes a vehicular control system architecture that supports self-configuration. The architecture is based on dynamic mapping of processes and services to resources to meet the challenges of future demanding use-scenarios in which systems must be flexible to exhibit context-aware behaviour and to permit customization. The architecture comprises a number of low-level services that will provide the required system functionalities, which include automatic discovery and incorporation of new devices, self-optimisation to best-use the processing, storage and communication resources available, and self-diagnostics. The benefits and challenges of dynamic configuration and the automatic inclusion of users' Consumer Electronic (CE) devices are briefly discussed and the self-management and control-theoretic technologies that will be used are described in outline. A number of generic use-cases have been identified, each with several specific use-case scenarios. To demonstrate the extent of the flexible reconfiguration facilitated by the architecture, some of these use-cases are described, each exemplifying a different aspect of dynamic reconfiguration.

Key words: Automotive Systems, Middleware Architecture, Dynamic Reconfiguration, Self-Configuring.

Anthony, R., Rettberg, A., Chen, D., Jahnich, I., de Boer, G., Ekelin, C, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 71–84.

1. INTRODUCTION

DySCAS (Dynamically Self-Configuring Automotive Systems) is a European Commission funded project that started in June 2006. DySCAS targets an automotive system that emphasizes the dynamic self-configuration of embedded systems. Unlike the common static configurations, the main goal of the project is the development of next-generation technologies based on existing solutions, namely the development of an intelligent automotive networked middleware system. This platform will have properties that include a high degree of robustness and fault-tolerance due to its self-adaptability at run-time. The cooperation of integrated system objects, primarily automotive ECU's and connected CE devices, results in a dynamic system that adapts itself to its changing environment and conditions.

2. WHY DYNAMIC RECONFIGURATION

The current state of practice is to embed several electronic control units (ECU) into a vehicle. These ECUs are typically based on proprietary hardware and software components and usually have specialized and fixed functionally. A vehicle may have several different ECUs, but since each one has a different non-transferable function, more devices means higher susceptibility to failure.

Upgrades are difficult and expensive; once a vehicle leaves the factory it is not easy to change its functionality, for example to apply the latest engine management configuration to improve fuel economy. If a serious fault occurs a manufacturer may have to undertake an expensive recall because vehicles cannot be easily upgraded in the field.

Owners have increasingly high expectations for infotainment services on the move, for example navigation devices are now common, and some treat their car as a mobile office. Therefore there is significant benefit to be gained by facilitating a three-way dynamic and automatic cooperation between consumer electronics devices that the owner brings into the vehicle, the embedded computing devices of the vehicle itself, and external networks that the vehicle passes within wireless range of.

Such a three-way setup enables an exiting and innovative suite of new applications that can: use location dependent information; share the processing power and other resources of several devices; achieve fault-tolerance by dynamic role allocation; and automatically collect and filter information (news, traffic, entertainment) in context-specific ways depending on who is in the vehicle, where they are and where they are travelling to.

A paradigm shift towards software-centric and network-centric ECU development is occurring. Software is becoming the dominant component of electronic automotive features. New opportunities are arising in the automotive electronics sector with networked and collaborating ECUs, which form an "automotive control grid" in which software components are no longer bound to specific hardware components, but can be executed on various units within the vehicles, and potentially even migrate during run-time. In this way each device can act as on-line replacement for other ECUs. In this scenario more devices means lower susceptibility to failure.

To maximise benefits from this opportunity, proprietary solutions will give way to standardised HW/SW component architectures and infrastructures. This provides developer freedom, better interoperability and service re-use, ultimately leading to lower development cost and higher reliability.

There is a lot of research into the migration of tasks from software to hardware. For example for hardware acceleration of image and video processing, specialized DSPs and FPGAs are used. Reconfigurable hardware/software based architectures are very attractive for implementation of run-time reconfigurable embedded systems. The hardware/software allocation of applications tasks to dynamically reconfigurable embedded systems (by means of task migration) allows for customization of their resources during run-time to meet the demands of executing applications, as can be seen in [1, 2]. DySCAS will focus mainly on the reconfiguration or self-configuration of software tasks in the middleware.

The AUTOSAR effort [3] has the purpose to provide an increasing flexibility with respect to the design time allocation of functions to different ECUs, by standardizing the software infrastructure and its services. Although this would be a large achievement, the AUTOSAR approach does not support dynamic system reconfigurations.

3. FUTURE VEHICULAR CONTROL SYSTEMS

The future vehicular control system takes the form of an embedded network system. This will comprise a wired core-network of processing nodes and sensor nodes spread throughout the vehicle. There will also be opportunities to incorporate mobile devices such as cell-phones, PDAs and other devices carried by the vehicle occupants. Such devices may have external connections to cellular networks and may also form ad-hoc networks between themselves. The automotive system will attempt to take advantage of the processing resources and connectivity of these devices when they are present.

4. SOFTWARE TECHNIQUES FOR SELF-CONFIGURATION

Context awareness and the ability to adapt behaviour to suit current environmental conditions are key requirements for projects such as DySAS. Configurational complexity arises from the number of components, the interactions between these components and between the components and their environment. This can be a barrier to achieving optimal or efficient performance and can be highly problematic for externally-manged run-time configuration [4, 5].

In answer to the complexity problem, the autonomic computing paradigm [6] advocates self-adaptation in which applications modify their behaviour to suit their environment and context, see for example [7] and [8]. There are many different types of adaptation [9], including self-configuration, self-healing, self-optimisation and self-protection [10, 11]; collectively referred to as 'self-management'. Self-management implies that the system is able to adjust some aspect of its own behaviour. When the adjustment is made to enable a new configuration to be supported, for example the dynamic incorporation of newly discovered devices the adjustment is termed 'self-configuration'. Adjustments to improve performance or efficiency are more-specifically referred to as self-optimisation. When the adjustment is made to mask or recover from a component failure it is referred to as self-healing. Adjustments made to deal with threats, such as dynamically detecting a denial-of-service attack by recognizing a 'pattern' in the requests to the system, are classed as self-protection.

The applications that populate the automotive system will have 'pervasive' and 'ambient' qualities. This concept involves embedding context-aware and location-aware services into infrastructure and portable devices. Ideally such services are so embedded that the user is not even aware of it [12]. To support this ideal, self-management moves the emphasis away from manual configuration of components, towards inbuilt learning and discovery capabilities [9, 10].

The Control-Theoretic approach for automatic control is receiving increasing attention [13] [14]. This approach has many characteristics which make it suitable for adoption within an autonomous computing framework: 1. Sensing of the system behaviour and model-based estimation of other interesting (and not measurable) system states; 2. Control based on feedback from sensed and estimated system states. Feedback control provides robustness to system disturbances and to uncertainties about the actual system behaviour; 3. Feedforward control, based on system models, to improve the system behaviour in response to known changes in the environment and changes of the desired behaviour; 4. Systems identification

and adaptive control as means to develop the models of the controlled environment, during design time and run-time.

Computer systems are traditionally modelled using e.g. queuing theory or other discrete-event formalisms. The dynamics differ where central ingredients include delays (due to queues) and dynamics are introduced through sensors. Despite this challenge, recent advances indicate that there is a lot to learn from control engineering. Combinations of feedforward and feedback control are useful and it seems that simple controllers are often sufficient.

Example application uses of a control approach include load balancing and QoS optimization. Typical application areas include multimedia, web storage systems, and network routers [15]. For embedded control systems, issues covered in the research include for example feedback-based resource scheduling, RTOS and middleware support for control activities, dynamic models of real-time systems, control and real-time co-design, and integration of quality-of-service and resource management [16].

5. A DYNAMICALLY RECONFIGURABLE ARCHITECTURE

The architecture needs to be capable of flexible and automatic reconfiguration in order to facilitate the technological advances targeted by the project. These advances include:

Automated fault detection, analysis and reporting: Trends in fault patterns can be learnt, and solutions devised. Over the longer-term solutions can be re-used.

Automated resilience through software relocation: Once a hardware or software fault has been identified it will be possible to automatically relocate the required functionality to another device.

Dynamic reconfiguration based on current resource demands: Processing power and storage capacity of ECUs and CE devices is typically quite limited. It will be possible to dynamically reconfigure resource provision and usage.

Software downloads of plug-and-play components: When passing through hot-spots, wireless connectivity to external systems will be automatically utilized to download new components and services, as well as for uploading fault status reports if necessary.

Automatic support for both push and pull software patching: The external connectivity permits automatic pushing of software revisions from the manufacturer side and faulty software needing replacement can be pulled by a specific system.

Sporadically available resources, as provided by a user's mobile device or accessible via a global network, will be included seamlessly into the vehicle's electronic system. The resources of a PDA or notebook could, for example, be used to improve the rendering of a 3D-view for navigation directions. The middleware needs to provide services that handle the availability and integration of such resources.

Simplification and standardisation of the software developer role: The provision of common interfaces to hardware and software components leads to a set of open standards. This avoids the complexity of having to deal with a multitude of different formats and thus removes the burden of interoperability from developers.

The core of the DySCAS architecture will be a self-managing middleware containing a number of services able to be dynamically composed to provide functionalities required by a wide range of use-cases (see next section).

Services identified so far include:

- Discovery (of location, of physical devices and of the services / capabilities provided by those devices),
- Interface provisioning / negotiation (brokering of service agreements between components),
- Resource mapping (dynamic determination of which resources should be allocated to which services),
- Security (authentication of devices offering or requesting services or resources),
- Storage management (especially when downloading SW; keeping track of versions),
- Rollback management (if a SW update fails, or an untested component configuration arises, a previously stable configuration must be reinstated),
- Reliable download (of SW, configuration parameters, data, or policy),
- SW / policy installer / upgrader (must ensure that the upgrade is performed at a safe moment, and if unsuccessful must invoke the rollback manager),
- Error management (dynamic fault detection, and on-board diagnostics),
- Migration of service (moving code and or data to balance load and to recover from faults),
- Data logging (sensor data is collated for subsequent diagnostics),

- Dynamic service prioritization (based on context, to support gradual degradation in the presence of HW or SW faults, battery-low condition etc.),
- SW / HW reconfiguration (low-level service to facilitate migration of service).

By providing these services the DySCAS middleware facilitates run-time reorganisation to balance workloads, expedite urgent processes, and to reconfigure components for survivability despite hardware failures. Components will also be able to automatically discover new components and establish service level agreements based on the resources and services that they are able to provide to each other.

The DySCAS architecture is specified in three stages: the System Architecture Framework (SAF), the System Architecture (SA) and the System Architecture Specification (SAS), see figure 1.

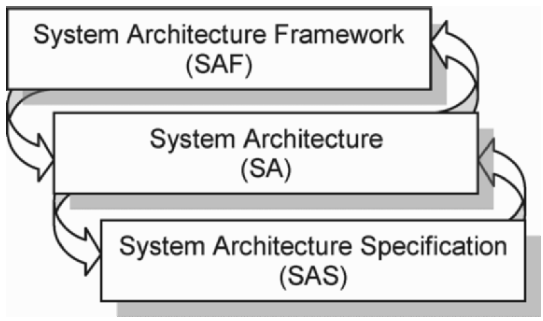


Figure 1. Specification of the DySCAS architecture.

The SAF identifies the system contents and provides the concept at a high-level. The list of system services is also included in SAF. Additionally, the SAF defines the notation for the SA and gives a description of the working process (i.e. how the components / services will interact, composition etc.). Together with the list of available technologies, the SAF defines the draft abstractions for the communication, the used operating system (OS), the underlying hardware resources, possible component models and software patterns. Therefore, the SAF covers all aspects of the implementation constraints.

The system structure (the applied APIs and components) are provided in the SA. Here protocol state machines are used for the description. This is more an abstract description of the functionalities and the APIs, but without a formal model. The formal model for this purpose is defined in SAS.

Components and APIs require an interface with attributes, properties and also timing information. This is also described in SA.

The SAS is as generic as possible to be open and easily adaptable and extendable. The specification described in an ADL provides the classification of services. Here the formal model for the system including an abstract description of the functionalities, APIs and the interfaces are given. Besides this, QoS assumptions and requirements for the used OS are defined. Therefore, the development of an abstract bus description, a so-called virtual function bus, including the communication properties to enable a design is necessary. To summarize, the SAS provides the user of the DySCAS architecture with guidelines for the implementation. This includes also the tool environment. Therefore, the SAS serves as documentation for application developers.

6. USE-CASES

The technologies under development within DySCAS are based on requirements derived from a rigorous analysis of future use-cases. A model describes three levels of use-case, in which generic use-cases represent groupings of specific use-cases requiring similar subsets of functionalities. These are atomic functionalities which can not be further broken down. Functionalities, in turn, map onto one or more services. A service may directly configure resources (in the technology layer) or may require further assistance from, or delegate to, other services. This structure (shown in figure 2) is important in the derivation of services and technology that form the DySCAS architecture, which has to be simultaneously very flexible and highly efficient.

One example of a system functionality is the function which enables a device (for example a smartphone) to connect to the car information systems. System functionalities are abstracted in the sense that they describe the functional requirements but not how they will be realized. The actual realization (below the horizontal line in figure 2) is done by defining services which will be implemented on top of the technical components.

Each of the use cases considered consists of a sequence of systems functionalities. When considering different use cases it was realised that groups of use cases exist which are each built up by a similar suite of system functionalities. To reflect this generic use cases are defined. The generic use cases identified so far are:

- New device automatically attached to the car,
- Integrating new software functionality,

- Closed re-configuration.

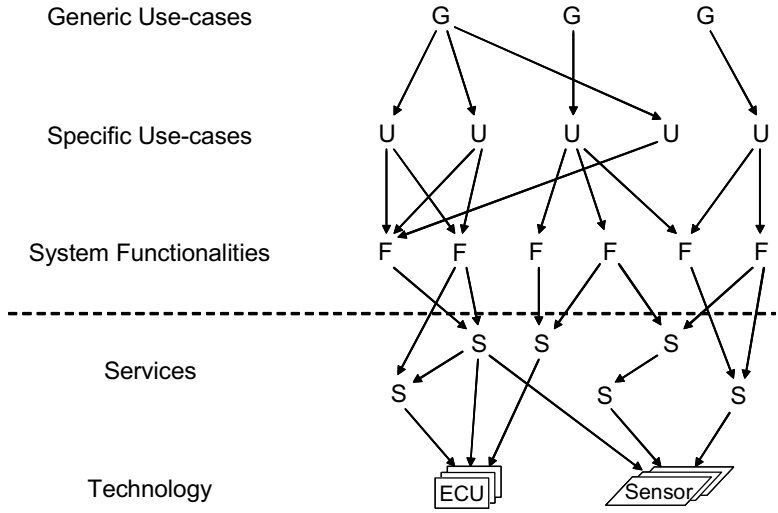


Figure 2. Mapping Use-cases onto DySCAS Services and Technology.

a) New device automatically attached to the car

If an appropriate new device is detected by the vehicle system it is automatically attached to the vehicle IT systems. This can be a mobile device brought into the car such as a PDA or smart phone, but it can also be a Hotspot outside of the car.

In these use-cases the information system of the vehicle is complemented by another device which may open access to external services, possibly extending the functionality of the system by increasing computing resources such as memory, computing capabilities etc, or it may act as a new source of data (e.g. an MP3 player). The task is to integrate the new device seamlessly and for the driver as transparently as possible.

Specific examples of the generic use-case include:

- Accessing a navigation system on mobile devices,
- Connecting a trailer and incorporating the computing and sensor systems the trailer is hosting,
- Internet access via an attached mobile phone (smart phone),

- Sending/receiving traffic data via an attached mobile phone,
- Streaming of entertainment data (MP3, video) from the mobile phone to the vehicle's entertainment system,
- Extracting the destination from the calendar application of the mobile device,
- Vehicle connecting to a wireless network hotspot,
- Determination of location, affecting context. Such as detecting the vehicle has entered the proximity of the owners workplace.

System functionalities include:

- SW is migrated to new HW,
- Wireless connection is established,
- Capabilities are negotiated,
- Authentication and all security mechanisms needed to protect the system.

b) Integrating new software functionality

This generic use-case covers all specific use-cases in which new software is included in the system. Software includes application software as well as operating software. The system must integrate the new software patches and ensure that the new configuration works properly. If not then the old software configuration must be reinstalled. This must be done as seamlessly as possible for the driver and it must not interrupt or impact in any way driving. Therefore the reconfiguration must be performed only in absence of safety related situations.

Examples of system functionalities are:

- Self-test,
- Security features,
- Reconfiguration.

Specific Use Cases include:

- Software download from hotspots,
- Software download from storage media (DVD),
- Replace system functionality,
- Replace applications (e.g. navigation, MP3 software players).

c) Closed re-configuration

In contrast to the other generic use-cases the closed re-configuration includes all use cases which perform any kind of reconfiguration which is not initiated by external events or by the integration of external hardware or software, but is performed to make the system work better or react to unexpected situations. To detect re-configuration opportunities, all components of the system must be monitored. Free devices or devices which do not show expected behaviour must be detected and well working configurations must be identified, migration must be planned and executed.

Specific use case examples are:

- Graceful degradation in case of power problems,
- Efficient usage of redundancy, e.g. ECUs which are used temporally, such as the (normally dedicated) security/immobilizer ECU, unused once the vehicle has been started.
- Migration of services in the case of HW failure.

System functionalities include:

- Detection, monitoring features,
- Re-configuraton features,
- Self-testing / diagnostics.

By means of an illustrated example, consider that an ECU failure is detected. The affected service must be migrated to an alternative ECU. If none are available, running services must be prioritised, so that an ECU can be made available by shutting down some services. This use-case is 'Migration of services in the case of HW failure' which is in the 'Closed re-configuration' generic class of use-cases. The composition of services needed to achieve the reconfiguration is illustrated in figure 3.

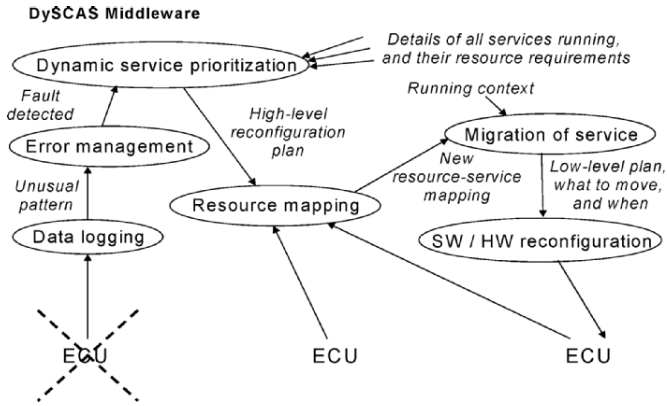


Figure 3. Composing DySCAS services to satisfy the 'Migration of services in the case of HW failure' use-case.

7. CONCLUSION

This paper comes at an early stage in the DySCAS project, and as such, it represents our preliminary design solution for self configuration. The project targets an ambitious collection of use-cases, requiring dynamic adaptation and thus the proposed middleware must provide a number of new services in a flexible and efficient manner.

Due to the timing of this paper, the architecture has been described at a high level. Some specific services have been identified and this set will stabilize as the use-cases are scrutinized during the next stage of the project. It is important that the services are clearly defined and differentiated to avoid duplication and to ensure that flexible composition of these services is possible.

Numerous use-cases have been identified at two levels, generic and specific. This approach facilitates identification of the required finctionalities, whilst once again avoiding duplication. Some of the use-cases have been discussed in outline.

8. NEXT STEPS

Now that the basic architecture has been determined, it needs to be refined. Further detail will be added by analyzing the use-cases in more detail and also by examining the features (capabilities, performance, resource constraints, interface issues) of the various platforms (ECUs, sensors etc.) and mapping the services onto these platforms.

A demonstrator application is under development; this will eventually showcase some of the use-cases in a real-time emulation. Further details of the project are available at the DySCAS website, see [17].

ACKNOWLEDGEMENTS

The authors would like to thank all the people and partners who are involved in the DySCAS project. Without their contributions and the fruitful discussions inside the project this paper could not have been written.

The DySCAS project is funded within the 6th framework programme "Information Society Technologies" of the European Commission.

REFERENCES

[1] Harkin, J., McGinnity, T. M., and Maguire, L. P. (2004). Modeling and optimizing run-time reconfiguration using evolutionary computation. Trans. on Embedded Computing Sys., 3(4):661–685.

[2] Götz, Marcelo, Rettberg, Achim, and Pereira, Carlos E. (2005). Towards Run-time Partitioning of a Real Time Operating System for Reconfigurable Systems on Chip. In Proceedings of International Embedded Systems Symposium - IESS, Manaus, Brazil.

[3] AUTOSAR – Automotive Open Systems Architecture. Available at http://www.autosar.org

[4] M. Ibrahim, R. Telford, P. Dini, P. Lorenz, N. Vidovic and R. Anthony, Self-adaptability and Man-in-the-Loop: A Dilemma in Autonomic Computing Systems, 2nd International Workshop on Self-Adaptable and Autonomic Computing Systems - SAACS '04 (DEXA 2004), Zaragoza, Spain, September 2004, IEEE.

[5] R. Barrett, P. Maglio, E. Kandogan and J. Bailey, "Usable Autonomic Computing Systems: the Administrator's Perspective", Proc 1st Intl Conf Autonomic Computing (ICAC 2004), IEEE Computer Society, New York, May 2004, pp. 18-25.

[6] Kephart J. O, Chess D.M. The Vision of Autonomic Computing. Computer, IEEE, Volume 36, Issue 1, January 2003, pp. 41-50.

[7] Dolev, Shlomi and Kat, Ronen I, Self-Stabilizing Distributed File Systems, International Workshop on Self-Repairing and Self-Configurable Distributed Systems on the 21st Symposium on Reliable Distributed Systems, October 2002, IEEE, pp. 384-390.

[8] N. Badr, A. Taleb-Bendiab, M. Randles, D. Reilly, 'A Deliberative Model for Self-Adaptation Middleware Using Architectural Dependency', Proc 15th Intl Conf Database and Expert Systems Applications (DEXA 2004), August 2004, IEEE, pp. 752-756.

[9] S. White, J. Hanson, I. Whalley, D. Chess and J. Kephart, An Architectural Approach to Autonomic Computing, Proc. 1st Intl. Conf. Autonomic Computing (ICAC), IEEE, New York, May 2004, 2-9.

[10] IBM, An architectural blueprint for autonomic computing, IBM Autonomic Computing White Paper, http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf

[11] J. Koehler, C Giblin, D. Gantenbein and R. Hauser, On Autonomic Computing Architectures, Research Report (Computer Science) RZ 3487(#99302), IBM Research (Zurich), 2003.

[12] Waldrop M, Autonomic Computing: The Technology of Self-Management, Woodrow Wilson International Center for Scholars, http://www.thefutureofcomputing .org/Autonom2.pdf, 2003.

[13] J.L. Hellerstein, Y Diao, S Parekh, D M Tilbury. Control engineering for computing systems - industry experience and research challenges. IEEE Control Systems Magazine, Volume 25, Issue 6, Dec. 2005.

[14] R Sanz, K-E Årzén. Trends in software and control. IEEE Control Systems Magazine, Volume 23, Issue 3, June 2003.

[15] K-E Årzén, A Cervin, T Abdelzaher, H Hjalmarsson, A Robertsson, Roadmap on Control of RealTime Computing System, EU/IST FP6 ARTIST2 NoE, Control for Embedded Systems Cluster.

[16] K-E Årzén, JPRA-NoE Integration: Adaptive Real-time, HRT and Control, Activity Progress Report for Year 1, IST-004527 ARTIST2 NoE: Embedded Systems Design.

[17] DySCAS website: www.dyscas.org

AN OSEK/VDX-BASED MULTI-JVM FOR AUTOMOTIVE APPLIANCES

Christian Wawersich, Michael Stilkerich, Wolfgang Schröder-Preikschat
University of Erlangen-Nuremberg
Distributed Systems and Operating Systems
Erlangen, Germany
E-Mail: wawi@cs.fau.de, stilkerich@cs.fau.de, wosch@cs.fau.de

Abstract: The automotive industry has recent ambitions to integrate multiple applications from different micro controllers on a single, more powerful micro controller. The outcome of this integration process is the loss of the physical isolation and a more complex monolithic software. Memory protection mechanisms need to be provided that allow for a safe co-existence of heterogeneous software from different vendors on the same hardware, in order to prevent the spreading of an error to the other applications on the controller and leaving an unclear responsibility situation.

With our prototype system KESO, we present a Java-based solution for robust and safe embedded real-time systems that does not require any hardware protection mechanisms. Based on an OSEK/VDX operating system, we offer a familiar system creation process to developers of embedded software and also provide the key benefits of Java to the embedded world.

To the best of our knowledge, we present the first Multi-JVM for OSEK/VDX operating systems. We report on our experiences in integrating Java and an embedded operating system with focus on the footprint and the real-time capabilities of the system.

Keywords: Java, Embedded Real-Time Systems, Automotive, Isolation, KESO

1. INTRODUCTION

Modern cars contain a multitude of micro controllers for a wide area of tasks, ranging from convenience features such as the supervision of the car's audio system to safety relevant functions such as assisting the braking system of the car. The diversity of hardware and software, likely to stem from a variety of manufacturers, complicates the integration to a connected and cooperating system.

Integrating multiple tasks on fewer, but more powerful micro controllers reduces diversity and costs of production. This approach, however, introduces

Wawersich, C., Stilkerich, M., Schröder-Preikschat, W, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 85–96.

new problems. In a system of dedicated micro controllers, the deployed software is physically isolated from each other. This isolation lacks among tasks sharing the hardware, which enables a malfunctioning task to corrupt the memory of other tasks, spreading the error and possibly resulting in a failure of other tasks running on the micro controller. The impact of such a failure depend on the duties assigned to the software and can be catastrophic. Software development in C and Assembler tends to be error-prone, yet they dominate the development of embedded software. This, along with the ever-increasing software complexity, even worsens the problem and accentuates the need for isolation concepts in embedded systems.

Micro controllers that are equipped with a memory protection unit (MPU) or a memory management unit (MMU) can isolate tasks, however, the hardware-based approach has several shortcomings compared to software-based protection. Beside these micro controllers being more expensive than controllers without an MPU/MMU, software development in a type-safe language such as Java avoids many errors that cannot be detected by unsafe languages, resulting in a more robust software. Moreover, hardware-based solutions are heterogeneous and require different programming on different hardware. Software-based solutions offer a uniform environment independent of the underlying hardware.

Though the approach of software integration arises the discussed problems of error-prone software development and lacking isolation, these problems are well-known from the area of personal computing and concepts have been developed to solve these problems. These concepts can be adapted and migrated to embedded systems. Object-oriented programming (OOP) facilitates coping with complex software and makes the software-development process more robust. Virtual machines such as the Java Virtual Machine (JVM) [9] allow to run applications in an isolated environment and provide memory protection through type-safety without the need for a MPU or even a MMU, preventing the use of arbitrary values as memory references.

Migrating these concepts to an embedded JVM [1, 11, 5, 4] provides less error-prone software-development using Java as an object-oriented programming language and isolates tasks (or threads, to speak in terms of Java) to a certain degree. Yet the approach of a single JVM has several shortcomings. First, static class fields allow the wandering of references. Second, resources are shared equally among the threads, which is not tolerable in real-time systems.

Our prototype system *KESO* will demonstrate that both, a robust development process for embedded software with OOP and the safe integration of tasks on a single micro controller by means of strong isolation, can be achieved while maintaining real-time capabilities and a small footprint suitable for the limited resources available on embedded systems. KESO is built on top of a standard

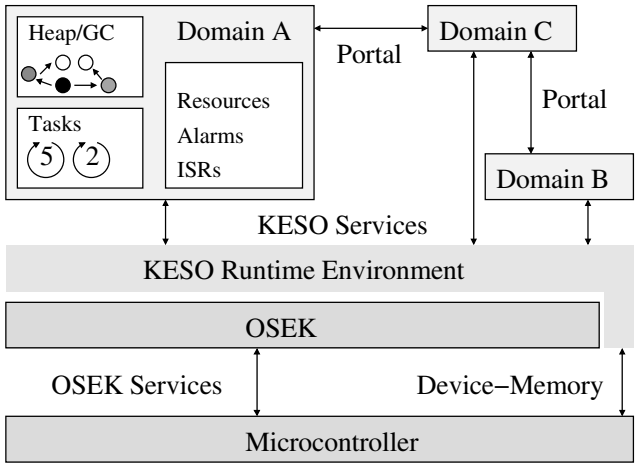


Figure 1. System Architecture

OSEK/VDX [10] operating system. These operating systems are widely used by the automotive industry. KESO thus provides a similar system configuration and creation process and a familiar interface to OSEK system services for embedded software engineers. We adopted the concept of isolated domains as it was first introduced by JX [8] and redesigned the architecture to fit more closely to the needs of resource-constrained environments.

The remainder of this paper is structured as follows: In section 2, we describe the overall KESO architecture and the build process. We evaluated our system with two example applications, which we describe in section 3. The paper is concluded with a discussion of related work in section 4.

2. KESO ARCHITECTURE

The architecture of KESO is illustrated in figure 1. The KESO runtime environment is based on top of an OSEK/VDX operating system, which affects the KESO design in several aspects.

Scheduling. OSEK/VDX operating systems use the concept of tasks as the basic schedulable unit. OSEK scheduling is based on static priorities. Consequently, KESO also uses the notion of priority-assigned tasks to represent threads of control rather than Java threads, whereby each task is assigned to a KESO domain. Scheduling is handled by the OSEK/VDX scheduler.

Synchronization. OSEK/VDX provides a synchronization mechanism that utilizes a priority ceiling protocol. We decided not to fully support Java

monitors because they would degrade the block-free properties of this protocol. We provide limited support for the Java *synchronize* primitive: Entering and leaving a monitor are mapped to acquiring and releasing the special OSEK scheduler resource, that has a ceiling priority equal or higher than the priority of the highest-priority task in the system. We do otherwise encourage the usage of the OSEK priority ceiling API that allows the complete exploitation of priority ceiling synchronization.

Domains. Each domain appears as a self-contained JVM to the user application and represents the fundamental unit of memory protection in a KESO system. The static class fields and the object heap of each domain are separated, whereby each domain can choose a management strategy for its heap.

Strong isolation of the domains is achieved by restricting the scope of each object to exactly one domain, the heap of which the object is allocated from. Each task belongs to exactly one domain. Upon a task switch the domain context is switched to the domain of the task, meaning that new objects are allocated from the heap of the respective domain and the task has a view on the static fields of its domain. This guarantees that a reference never crosses a domain boundary.

Domains are also the fundamental unit of allocation of the resource memory, in what domains differ from Java isolates [2] as used in Squawk VM [12].

Portals. Inter-domain communication is possible via portals. A domain can provide a *portal service*. A portal service consists of a Java interface that offers *service methods* to other domains. The service class that provides the implementation of the portal interface is exported via a static global name. Both the class and the name are configured in the system configuration. Tasks of other domains (*client tasks*) can invoke methods of the portal. The execution of a service method takes place in the environment of the service domain.

The parameters of a portal call are passed by value. If an object is passed as a parameter to a service method, a copy of the object is allocated on the heap of the service domain. Thereby the property of the separated heaps is preserved. The service is executed by the calling OSEK task, which is migrated to the service domain for the duration of the service. The code executed by the migrated task will execute with the scope of the service domain, i.e. has the same view on the system as any regular task of the service domain.

This solution has the advantage that the service method is executed with the priority of the calling task. This conforms to the OSEK priority model. The alternate solution of a dedicated service task in the service domain would require a service task of the same priority as the client task. Thus a service task would have to be created for every possible client task to conform to the priority model. This would use reasonable more resources than the task migration.

OSEK/VDX API. The OSEK/VDX API of KESO [13] allows the user applications to access OSEK services such as the scheduler and the synchronization primitives. This API does also provide mechanisms that allow to restrict the access to the system services to a particular domain in order to guarantee strong isolation.

Hardware Access. Embedded applications often require access to the hardware. In KESO, access to memory mapped device registers is possible through the *Memory* interface. This interface provides methods to access a specific region of memory with methods similar to raw memory access. The memory region accessible through a Memory object can be bounded to prevent a breakout from the Java protection mechanisms. Port-based access to hardware is possible through a similar interface.

These interfaces allow the implementation of the device drivers in Java. No references can be stored through this interface, which prevents the transition of a reference through a commonly accessed region of device memory. Memory objects do also provide a way to efficiently share large amounts of data between different domains without the need to copy.

Heap Management. KESO currently provides three different flavors of heap management: no garbage collection at all, a stop-the-world garbage collector and a highly preemptable garbage collector (RTGC). Each domain can choose an appropriate strategy depending on its needs, which is possible due to the strict separation of the domain heaps. The real-time specification for Java (RTSJ) [6] addresses a similar topic with the introduction of immortal and scoped memory, however, the handling of object references of different heap types in the RTSJ is costly why we opted for a clear separation.

Code Generation Concept. The user applications are developed in Java and available as Java bytecode after having been processed by a Java compiler. The bytecode is compiled to C source code ahead of time by the *KESO builder*. The generation process of a KESO system is illustrated in figure 2.

The generated C code does not only contain the compiled class files, but also the KESO runtime data structures. Moreover, additional code is inserted to retain the properties of a JVM, such as `null` reference checks and array boundary checks, and the code of other services of the KESO runtime layer, such as the garbage collector and the portal services.

Code Size Optimizations. KESO is a static system, that does not allow the dynamic loading of classes at runtime, which opens more optimization potential for reducing the size of the generated system.

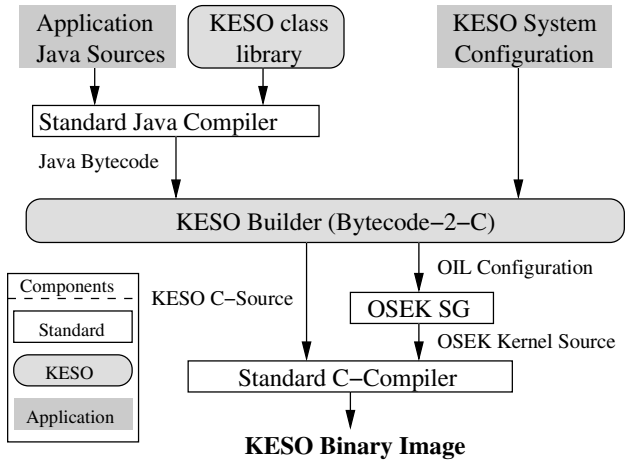


Figure 2. System Generation Process

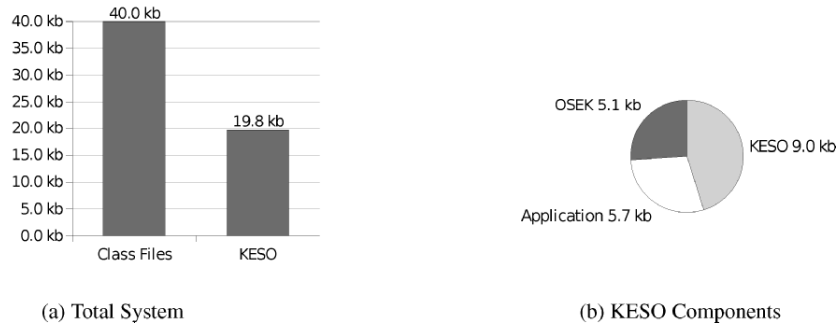
The KESO compiler performs a reachability analysis on the bytecode and eliminates classes, methods and fields, that are not accessed by the user application. This reduces both, the code size of the generated system and the size of the KESO data structures. Additionally, only the parts of the KESO abstraction layer, that are actually used by the application, are added to the generated code, e.g. if an application does not use OSEK resources, the data structures and code for the resource-related services are not added to the generated system.

The data structures of the runtime environment are automatically generated for the smallest memory footprint. The class type information only consumes a few bytes per class, depending on the number of classes, and can be moved completely into the ROM or flash memory. Virtual method calls are eliminated by the compiler as far as possible and the remaining virtual method calls can be entirely replaced by conditional branches.

While the KESO builder is performing global optimization, the generated C source code is also enhanced with additional compiler hints for better code generation. For instance, the C compiler can perform common sub-expression elimination with C functions if the return value of a function only depends on its arguments and the function does not access global memory.

3. EVALUATION

Garbage Collector. Figure 3 shows a footprint comparison for a test application that was used to test the real-time garbage collector. The test application processes an infinite loop and maintains a FIFO of fixed size. In each



(a) Total System

(b) KESO Components

Figure 3. Footprint comparison of the garbage collector test application

loop cycle, a new element is allocated and added to the FIFO, and another element is removed. The task is activated by an OSEK event triggered by a cyclic OSEK alarm after each loop cycle. The garbage collector can reclaim memory in the waiting phases of the task. As the task enters the waiting state, the garbage collector has to scan the stack of the task. OSEK resources are used to synchronize the stack scanning with the task. Hence the test application requires an almost full-featured OSEK operating system. The applications makes extensive used of string manipulation operations, which is the main reason for the large code size, but was convenient because a lot of memory is allocated during these operations.

Figure 3(a) opposes the size of the class files actually used by the test application and the size of the resulting KESO image. The KESO image is only 50% of the size of the class files, and already includes the KESO VM and the OSEK OS. Looking at the portion the optimized and compiled files take in the binary image (Figure 3(b)), the 40 kB of class files are compiled to only 5.7 kB which is 14.25% of the original size. For comparison, Squawk VM [12] achieves a size reduction to 38% on average by converting class files to an internal suite file format.

Figure 3(b) shows the composition of the KESO system. The bulk of the system is posed by the KESO runtime, that consists almost half (4 kB) of the code of the garbage collector that will remain constant for larger applications. The other major part of the KESO layer is posed by the runtime data structures. These data structures grow with the number of classes and methods used in the system. The size of the OSEK system also depends on the needs of the user applications, however, the test application already makes use of OSEK resources, alarms and events. Thereby, even for larger applications, only a decent increase of the OSEK component has to be expected. The most variable

component is posed by the user application. With larger applications, the application component will increase in size most, and the fraction of the other two components in the size of the entire system will shrink.

The RTGC disables the interrupts in several critical sections. As this can delay the reaction to external events, it is important to determine the worst-case latency that can be caused by the RTGC. All critical sections that are secured by disabling the interrupts are O(1). We measured an execution time of 8 μs for the longest critical section of the RTGC on a Tricore controller clocked at 150 MHz. This is less than the time spent in the critical section of the frequently used `ActivateTask()` service of the OSEK OS (12 μs). Our RTGC does therefore not increase the worst-case latency to events.

Small Real-World Application. For a first evaluation of our system with a real world application, we ported an application that was created by students as an exercise to a local real-time systems lecture to KESO. The original application is written in C and runs on top of the same OSEK/VDX implementation that is used for KESO.

The application is a control software to an automated version of an experiment similar to the ring-the-bell game. Figure 4(a) shows the schematic construction. An iron projectile can be raised and lowered in a plastic pipe using seven electric coils and gravity. The coils are mounted in spacings of 230 mm on the pipe. A photo sensor is attached to each of the coils to detect the movement of the projectile within the pipe. Both the photo sensors and the coils are connected to the general purpose I/O ports of a an Infineon Tricore TC1796b micro controller.

The application implements a finite state machine (FSM). State transition is either caused by interrupts of the photo sensors or a timer (using an OSEK alarm), depending on the state of the FSM. The experiment represents a hard real-time problem. If the application fails to (de)activate a coil in order to catch or decelerate a fast ascending projectile, the projectile will be shot out of the pipe and the programmed sequence cannot be completed any more.

We determined a maximum latency of 240 μs to react to the interrupt of a photo sensor. The successful execution of the program cannot be guaranteed with higher latencies anymore. The measured interrupt frequency ranges from 10 Hz to 27 Hz.

The ported application is implemented very similar to the original C application to ensure comparability. The application does only allocate memory in the startup phase and does therefore not require a garbage collector. As there is only a single OSEK task, the KESO system was configured with a single domain only.

Figure 4(b) shows a comparison between the used Java classes, the original C code and the different configurations of the KESO system. The jar file only

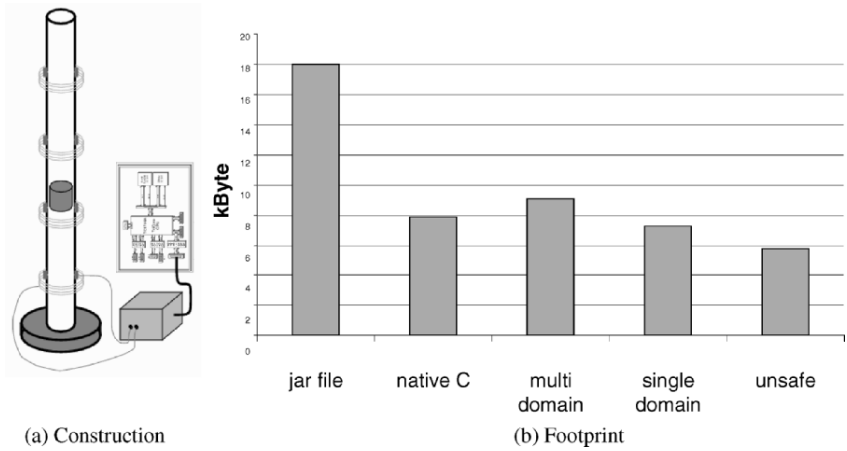


Figure 4. Ring-the-bell Simulator

contains the classes which are actually used by the program and have to be loaded by the JVM. The original C application has a code size of 7.9 kB and a multi domain KESO configuration has 9.1 kB. We expect that the overhead is growing slower than the code size of bigger applications.

The evaluated application does not require multi domain support. The single domain configuration with only 7.3 kB code size is thus the comparable image. The smaller code size was achieved by dead code elimination and auto generated compiler hints for C compiler. This compensated the extra runtime checks added for null pointer and boundary checks. An unsafe system configuration without the runtime checks has only a code size of 5.8 kB, which serves as a basis of comparison to see the amount of code eliminated by the reachability analysis and compiler hints.

4. RELATED WORK

The AJACS Project. The AJACS (Applying Java to Automotive Control Systems) [1] project did general research on deploying Java in automotive control systems, i.e. on static embedded systems, and therefore has the same target platforms as the KESO system. The main objective was developing and defining an open technology that is based on existing standards of the automotive industry, explicitly naming OSEK/VDX operating systems. The expected benefits of using Java on automotive control systems were restricted to a single JVM approach, particularly to software structuring, reusability, dependability, portability and robustness benefits. KESO mainly differs from AJACS in the

Multi-JVM approach, that puts the main focus on the isolation of the tasks integrated on the controller.

Squawk VM. The Squawk VM [12] is a small JVM written in Java that runs without an operating system. Java bytecode is converted to a Squawk specific *Suite File* format, that incorporates space, execution and garbage collection simplification optimizations.

Squawk implements an isolation mechanism similar to that of Java Specification Request (JSR) 121 [7, 2]. Squawk encapsulates applications into so-called *Isolates*, whereby each Isolate maintains an own copy of mutable data such as static class variables. An Isolate may contain multiple threads, therefore the Isolate concept shows some similarities to the domain concept used in KESO. Contrary to domains, all Isolates allocate new objects from the same heap, therefore Isolates are no separate units of memory allocation.

Squawk further differs from KESO in the thread and scheduling concept, and the non-preemptable system code including the garbage collector, that can have a negative impact on the interrupt handling latency.

JamaicaVM. The commercial JVM JamaicaVM [11] is designed as a base for embedded software that provides support for the RTSJ [6]. The Jamaica VM build tools also generate native code ahead of time. The main objective of Jamaica VM is to remove the non-determinism from Java in order to create a JVM that is suitable for hard real-time purposes. Jamaica VM as a JVM implementation does not offer concepts for thread isolation.

AUTOSAR. AUTOSAR [3] is a joint effort of the automotive industry to develop a standardized software infrastructure and the specification of compatible functional interfaces. The goal of the initiative is the replacement of the component-oriented development process by a function-oriented process.

AUTOSAR applications need to be specified as *software components* that can be transparently mapped to electronic controller units (ECU) in a network of ECUs. To enable this, AUTOSAR provides a *basic software* that abstracts from the hardware by providing a comprehensive set of drivers and hardware abstraction layers. Components communicate through the *virtual functional bus*. The communication over this bus is uniform regardless whether the communicating software components are placed on the same ECU or on different ECUs of the network.

Though outside the scope of this paper, KESO pursuits similar objectives. In an earlier paper [14], we presented a variant of KESO where domains can be distributed across ECUs in a network transparently to the application and communicate using the uniform portal mechanism. Contrary to KESO,

AUTOSAR does not provide the benefits of a type-safe language that allows the safe integration of components on an ECU.

5. CONCLUSION

KESO is the first embedded Multi-JVM for the automotive environment. It eases the transition from dedicated micro controllers to the integration of multiple applications on the same micro controller providing application isolation on micro controllers without the need of hardware-based memory protection. Hardware-based memory protection tends to more expensive hardware and the existing heterogeneous hardware solutions leading to a more complicated software integration process. With Java, we offer a robust and comfortable software development process.

We decided to use an OSEK/VDX system as the base of our development. OSEK/VDX systems are established in the automotive area and known for their small code size and good real-time properties. In KESO, the OSEK/VDX concepts were preferred over the Java compatibility to conserve these properties and to provide a familiar API to the software developer.

While KESO also offers powerful features such as garbage collection, these can be disabled for the entire system or only parts of the system, so that the overhead introduced by these features is only added to the system where required. Our powerful tools mostly automatically generate a system that is closely fitted to the needs of the applications.

The small controller application that we ported from C to KESO shows, that it is possible to produce similar code size in spite of the fact that we added runtime type and boundary checks. This was achieved because of the type-safe Java bytecode, which is easier to analyze and therefore has better attributes for global optimizations.

We are currently working on device drivers for KESO that allow a more generic and high-level access to the hardware and greatly increase the portability of KESO applications. In the future, we hope to provide a common runtime environment to embedded software regardless of the underlying controller hardware. For software that was already certified for one micro controller, this will hopefully ease or even obsolete the certification process of the same software for other micro controllers.

REFERENCES

[1] AJACS: Applying Java to Automotive Control Systems Concluding Paper V2.0. Technical report, AJACS consortium, July 2002. http://www.ajacs.org/.

[2] JSR 121 Application Isolation API Specification. Technical report, Palo Alto, CA, USA, June 2006. http://jcp.org/aboutJava/communityprocess/final/jsr121/.

[3] AUTOSAR. AUTOSAR - current results and preparations for exploitation. In *Proceedings of the 7th EUROFORUM conference: Software in the Vehicle*, 2006.

[4] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbruster, E. Pla, and D. Holmes. A real-time java virtual machine for avionics - an experience report. In *IEEE Real Time Technology and Applications Symposium*, pages 384–396, Washington, DC, USA, 2006. IEEE.

[5] D. Beuche, L. Büttner, D. Mahrenholz, W. Schröder-Preikschat, and F. Schön. JPure - purified java execution environment for controller networks. In *International IFIP TC10 W'shop on Distributed and Parallel Embedded Systems (DIPES '00)*. Kluwer, Oct. 2000.

[6] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. AW, 1st edition, Jan. 2000.

[7] G. Czajkowski. Application isolation in the Java virtual machine. In *15th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '00)*, pages 354–366, New York, NY, USA, 2000. ACM.

[8] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX operating system. In *2002 USENIX TC*, pages 45–58, Berkeley, CA, USA, June 2002. USENIX.

[9] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. AW, second edition, 1999.

[10] OSEK/VDX Group. *Operating System Specification 2.2.3*. OSEK/VDX Group, Feb. 2005. http://www.osek-vdx.org/.

[11] F. Siebert. The impact of realtime garbage collection on realtime Java programming. *isorc*, 00:33–40, 2004.

[12] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™ on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *2nd USENIX Int. Conf. on Virtual Execution Environments (VEE '05)*, pages 78–88, New York, NY, USA, 2006. ACM.

[13] M. Stilkerich, C. Wawersich, W. Schröder-Preikschat, A. Gal, and M. Franz. OSEK/VDX API for Java. In *Linguistic Support for Modern Operating Systems ASPLOS XII Workshop (PLOS '06)*, pages 13–17, San Jose, California, USA, Oct. 2006. ACM.

[14] C. Wawersich, M. Stilkerich, R. Ellner, and W. Schröder-Preikschat. A Distributed Middleware for Automotive Applications. In *Proceedings of the 1st Workshop on Models and Analysis for Automotive Systems*, pages 25–28, Dec. 2006.

TOWARDS DYNAMIC LOAD BALANCING FOR DISTRIBUTED EMBEDDED AUTOMOTIVE SYSTEMS

Isabell Jahnich and Achim Rettberg
University of Paderborn/C-LAB, Germany
{isabell.jahnich, achim.rettberg}@c-lab.de

Abstract: This paper describes a middleware architecture for distributed automotive systems that supports dynamic load balancing of tasks. Load balancing could be applied if an external device is added to the vehicle. The middleware architecture has to deal with the fusion of such non-built-in devices. An important factor to set up such a system is the identification of the requirements that should be handled by the middleware architecture. Enriching the middleware with traditional load balancing strategies allows an easy exchange of tasks.

Keywords: Dynamic Load Balancing, Automotive Systems, Middleware Architecture

1. INTRODUCTION

Future application scenarios for vehicle electronic systems include the access to mobile devices that build ad-hoc networks with the built-in devices of the vehicle. Computer power and services of connected external devices to the built-in vehicle system allow the realization of new resource-intensive and innovative applications.

Nowadays powerful mobile consumer electronic devices like mobile phones and PDAs are very popular. Their connection to the vehicle system allows the migration of resource-intensive applications or tasks to increase the driver's comfort by faster processing, and to offer additional services.

To realize a seamless connection of mobile consumer electronic devices to the vehicle infrastructure, a middleware architecture is needed that supports the fusion of the additional device and the vehicle. Thus the attachment and the detachment of non-built-in devices have to be discovered and detailed device information and resources have to be registered. Furthermore the architecture is also responsible for the possible load balancing by migrating tasks from the vehicle system to the additional device.

Jahnich, I., Rettberg, A, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 97–106.

In the following a vehicle middleware architecture is presented that supports the attachment of additional devices. Furthermore it offers services to realize a load balancing based on different strategies. The rest of the paper is organized as follows: Section 2 will describe the related work in the field of research where our architectural approach is located. As a motivation for this paper Section 3 describes a use case scenario followed by the requirements of the architecture that is addressed in Section 4. Section 5 explains our architecture which is finally mapped to the use case scenario of Section 3. Some possible strategies of load balancing are regarded in Section 6 and finally Section 7 will give a conclusion and future work motivations.

2. RELATED WORK

There are several publications regarding load balancing and extensive research has been done on static and dynamic strategies and algorithms [6].

On the one hand, load balancing is a topic in the area of parallel and grid computing, where dynamic and static algorithms are used for optimization of the simultaneous task execution on multiple processors. Cybenko addresses the dynamic load balancing for distributed memory multiprocessors [3]. In [5] Hu et. al. regard an optimal dynamic algorithm and Azar discusses on-line load blancing [5]. Moreover Diekmann et. al. differentiate between dynamic and static strategies for distributed memory machines [4]. Heiss and Schmitz introduce the *Particle Approach* that deals with the problem of mapping tasks to processor nodes at run-time in multiprogrammed multicomputer systems solved by considering tasks as particles acted upon by forces.

All these approaches have the goal of optimizing the load balancing in the area of parallel and grid computing by migrating tasks between different processors, while our approach focuses the direct migration of selected tasks to a newly added resource. Furthermore we regard load balancing that is located on the middleware-layer.

Moreover there are static approaches, like [11], that address a finite set of jobs, operations and machines, while our approach deals with a dynamic set of tasks and processors within the vehicle system.

Balasubramanian, Schmidt, Dowdy, and Othman consider in [7], [9], and [8] middleware load balancing strategies and adaptive load balancing services. They introduce the *Cygnus*, an adaptive load balancing/monitoring service based on CORBA middleware standard. Their concept is primarily described on the basis of a single centralized server, while decentralized servers that collectively form a single logical load balancer is not explained in detail.

Moreover the topic of dynamic reconfigurable automotive systems is regarded in [2] and [1].

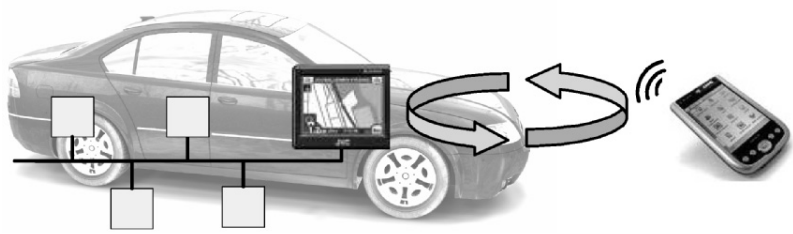


Figure 1. Use case scenario.

3. MOTIVATION

To run applications or tasks for example of the vehicle infotainment system more efficiently, a migration to the additional mobile device makes sense to use its unused resources. Thus it is possible to migrate for example tasks of the navigation system to the connected PDA for faster and more detailed map rendering and more optimal calculation of routing information.

After the connection of the PDA to the vehicle infotainment network with the aid of standardized interfaces like Bluetooth or WLAN, the device is discovered and the appropriate device information, locally running processes, and device and network resources are registered by a dedicated service.

In consideration of all running processes and the resources situation within the vehicle infotainment system appropriate services decide on a possible load balancing according to different strategies and initiate the task migration where required. Thus the appropriate navigation system tasks migrate from the navigation system to the PDA. And after the calculation the results of the tasks are sent back to the navigation system, where they are used.

4. REQUIREMENTS OF THE ARCHITECTURE

To realize the use case scenario described above a middleware architecture is required that fulfills several requirements.

- Event management
  - Additional devices have to be discovered by the vehicle infrastructure.
  - Device removals have to be discovered by the vehicle infrastructure.
- Device registration
  - Detailed information and capabilities of the newly added devices have to be registered.

- Resource management
  - Status information and resource load of each device of the vehicle have to be known.
- Load balancing
  - Potential task migrations have to be initiated based on different strategies.

5. ARCHITECTURE

In this section, we will give an overview of the proposed architecture. The architecture fulfills the requirements described in the previous section.

The operating system builds the interface between the hardware and the middleware (see Figure 2). Additionally, device drivers are necessary for specific hardware parts. The tasks run on top of the middleware. Middleware is a software layer that connects and manages application components running on distributed hosts. It exists between network operating systems and application components. The middleware hides and abstracts many of the complex details of distributed programming from application developers. Specifically, it deals with network communication, coordination, reliability, scalability, and heterogeneity. By virtue of middleware, application developers can be freed from these complexities and can focus on the application's own functional requirements.

Before explaining the design of our automotive middleware and the specific services, we enumerate the five requirements of automotive middleware. These requirements are resource management, fault-tolerance, specialized communication model for automotive networks, global time base, and resource frugality. These requirements are derived from the distributed, real-time, and mission-critical nature of automotive systems and differentiate automotive middleware from conventional enterprise middleware products.

A vehicle has a real-time nature. It is a system in which its correctness depends not only on the correctness of the logical result, but also on the result delivery time. Since a vehicle is subject to various timing constraints, every component in a vehicle should be designed in a way that its timing constraints are guaranteed a priori. At the same time, the timing constraints of a vehicle should be guaranteed in an end-to-end manner since an automobile is a distributed system and its timing constraints are usually specified across several nodes. For example, let us consider a typical timing constraint of an automobile. If pressing a brake pedal is detected at the sensor node, then the brake actuator node must respond to it within 1 ms. To meet this constraint, there must be a global resource manager that calculates the required amount of resources on each node and actually makes resource reservations to network

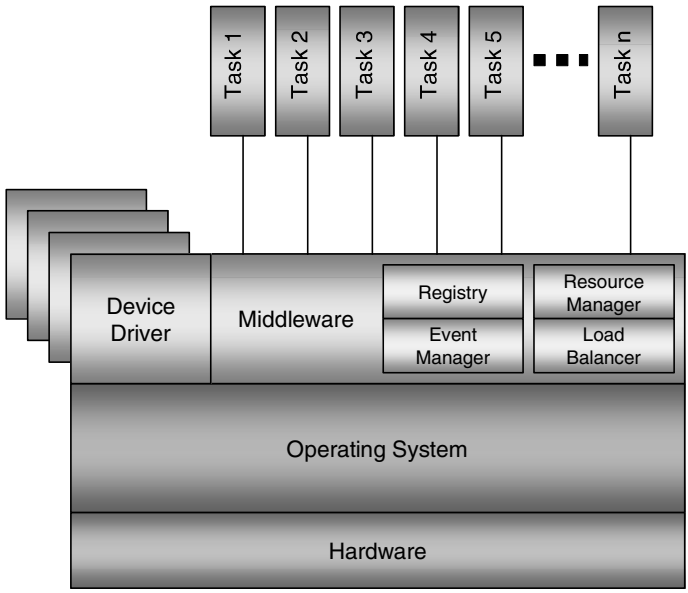


Figure 2. Self-configurable Architecture.

interface controllers and operating systems on distributed nodes. Automotive middleware is responsible for such resource management.

The middleware in our approach includes four components that offer specific services: *Registry*, *Event Manager*, *Resource Manager* and *Load Balancer* (see Figure 2).

The Event Manager is responsible for the device discovery. If a new device is added to the automotive system via technologies like Bluetooth or WLAN for example, it is recognized by the Event Manager component. Vice versa the Event Manager also notices the detaching of the device. In both cases it will inform the Registry of the middleware about the availability or the detaching of the additional device.

New devices are registered and detached devices are unsubscribed within the Registry service. During the registration the specific characteristics of the device (like memory, CPU, etc.) are stored within the Registry. Due to the distributed system the Registries of each vehicle ECU (Electronic Control Unit) communicate with each other to guarantee that each Registry of an ECU knows the actual status of all devices within the network inclusive of the newly added devices.

The Load Balancer spread tasks between the vehicle ECUs in order to get optimal resource utilization and decrease computing time. It evaluates possible migration of tasks based on different load balancing strategies. To guarantee a suitable migration the Load Balancer considers the current resource situation

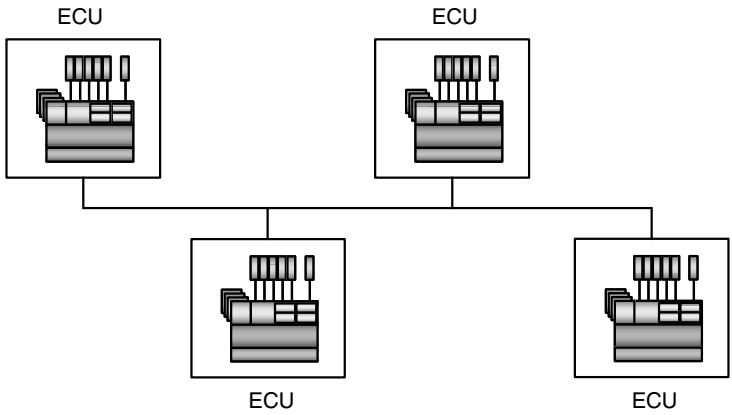


Figure 3. Distributed architecture.

on the ECUs with aid of the Resource Manager. Once a load balancing on an additional device is started, and this device is detached while the migrated tasks are executed, they will be re-started on the original ECU again. In this case the Event Manager is responsible to inform the Load Balancer to initiate this re-start.

The Resource Manager supervises the resources of the local ECU. To be aware of the complete network resource situation all Resource Managers synchronize with each other. Thus the Load Balancer gets the current resource situation of the complete vehicle infrastructure with aid of its local Resource Manager.

In our approach, the middleware is located on each ECU in the vehicle (see Figure 3). Every ECU has a unique ID. The ECU with the lowest ID is the master. Thus it is responsible for the control of the entire vehicle network, and newly connected and the detaching of additional devices are discovered by its Event Manager, device information is registered by its Registry, and its Load Balancer is responsible for the evaluation of the possible migration with the aid of the local Resource Manager. If the master ECU fails a new master will be chosen with the aid of the Bully-Algorithm [10].

5.1 SERVICE COMMUNICATION

According to the use case scenario description of Section 3 the connection of a mobile device like a mobile phone or PDA to the vehicle infrastructure is followed by several actions and communications (see Figure 4). In the following these activities will be regarded:

After the PDA has been attached to the vehicle infrastructure, the Event Manager of the master ECU recognizes the attachment of the additional device

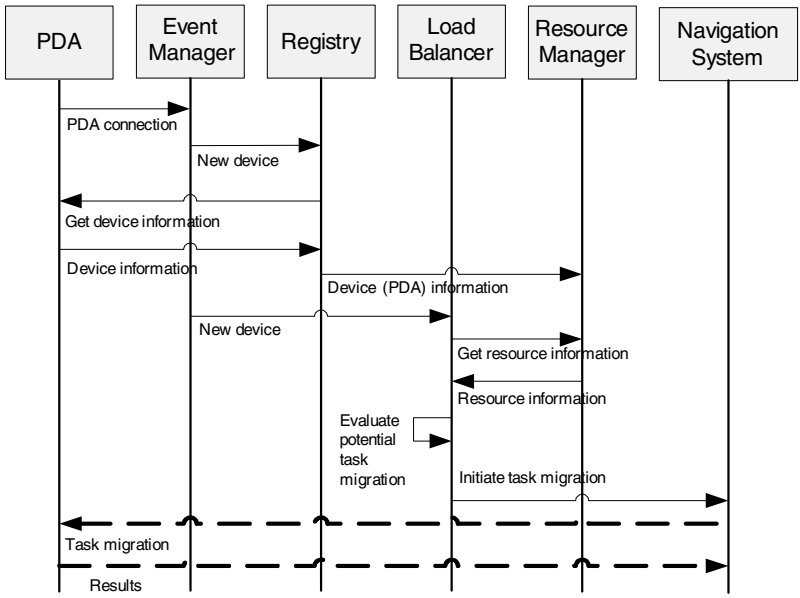


Figure 4. New device connected to the system.

and informs the local Registry about the newly added node. In the following this Registry asks the PDA for device and actual resource information. This information is stored by the Registry. Furthermore the resource details are forwarded to the local Resource Manager. The communication between the Registries of all ECUs makes sure that every ECU is informed about the additional device.

The event of the added PDA is also forwarded to the local Load Balancer by the Event Manager. To decide for a suitable load balancing strategy (see Section 6) it needs resource information about the current status within the network with aid of the Resource Manager.

According to the use case description of Section 3 the navigation system tasks migrate from the navigation system to the PDA and the results are sent back after the calculation.

6. LOAD BALANCING STRATEGIES

There are several possibilities to balance the load after additional devices have been connected to the vehicle network. Initiated by the Load Balancer component the new resources can be used and applications or tasks can be migrated to the additional device.

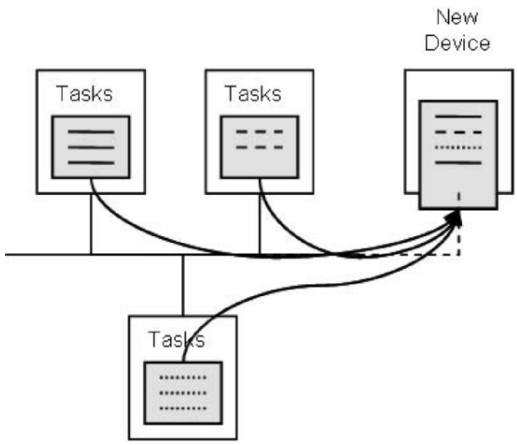


Figure 5. Round-Robin.

In the following three strategies will be explained: Task migration based on the Round-Robin algorithm, a strategy based on a migration from the most loaded processor and a cost based algorithm.

6.1 ROUND-ROBIN

As one of the simplest scheduling algorithms for processes *Round-Robin* assigns time slices to each process in equal portions and in order. All processes are handled without priority.

Based on this idea the Load Balancer assigns tasks to the processor of the additional device for a dedicated time slice. These tasks are migrated to the connected device and all resources of the vehicle infotainment network are relieved in an even manner (see Figure 5).

6.2 MOST LOADED

With the aid of the Load Monitor the current load of all nodes within the vehicle infotainment network will be monitored. The Load Balancer can request that information to decide whether to relieve busy devices by initiating a migration of tasks to the newly connected device. For this it generates a priority list which ranks the tasks from the busiest processor. In that way the tasks with the highest priority will be migrated to the resources of the additional device (see Figure 6).

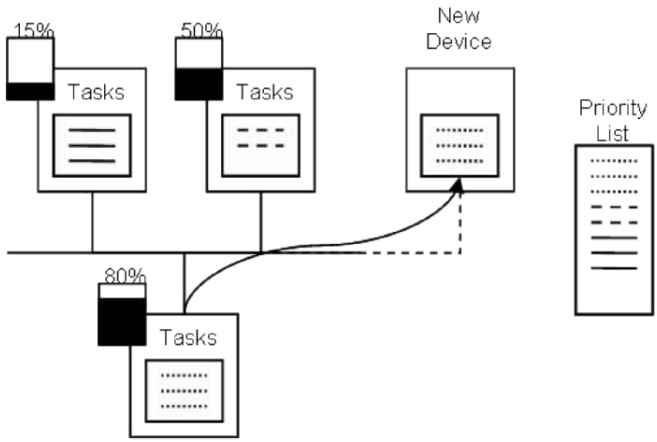


Figure 6. Most Loaded.

6.3 COST BASED MIGRATION

Within the cost based strategy the Load Balancer evaluates possible migration of tasks to the additional device. Migration is only a useful option if the cost of migrating is lower than the cost of keeping tasks with their original device. The cost benefit ratio for tasks of busy devices is computed which helps the Load Balancer to form the decision of whether to migrate or not. The calculation of migration costs of task is realized according to the priority list of the *Most Loaded* strategy.

7. CONCLUSION AND OUTLOOK

We presented a middleware architecture for automotive systems that enables dynamic load balancing. The integration of load balancing is a step towards a self-configuration within the vehicle. We focus on a specific use case scenario whereby an external device like PDA or mobile phone is added to the vehicle. With the help of the requirements, we described the middleware architecture and their enrichment with new services to support the distribution and exchange of tasks. Furthermore, we describe how traditional load balancing strategies could be applied within our approach.

Future work will be a detailed evaluation of the already existing load balancing strategies in the context of automotive systems. Additionally, the extension of existing or the development of new load balancing strategies will be done together with the implementation of the proposed architecture.

ACKNOWLEDGMENTS

This project was funded by the EU Commission within the project DySCAS (**Dy**namically **S**elf-**C**onfiguring **A**utomotive **S**ystems).

REFERENCES

[1] R. Athony, C. Ekelin, D. Chen, M. Törngren, G. de Boer, I. Jahnich, and et. al. A future dynamically reconfigurable automotive software system. In *Proceedings of the "Elektronik im Kraftfahrzeug"*, Dresden, Germany, 2006. Springer.

[2] R. Athony, A. Rettberg, I. Jahnich, C. Ekelin, and et.al. Towards a dynamically reconfigurable automotive control system architecture. In *A.Rettberg, R.Dömer, M.Zanella, A.Gerstlauer, F.Rammig. Proceedings of the IESS'07*, Irvine, California, USA, 2007. Springer.

[3] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. In *J. Parallel Distrib. Comput.*, 1989.

[4] R. Diekmann, B. Monien, and R. Preis. Load balancing strategies for distributed memory machines. In *In H. Satz F. Karsch, B. Monien, editor, Multiscale Phenomena and Their Simulation. World Scientific.*, pages 255–266, 1997.

[5] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. In *citeseer.ist.psu.edu/hu95optimal.html*, volume DL-P-95-011, 1995.

[6] Chi-Chung Hui and Samuel T. Chanson. Improved strategies for dynamic load balancing. In *IEEE Concurrency*, 1999.

[7] Balasubramanian Jaiganesh and Douglas. Evaluating the performance of middleware load balancing strategies. In *citeseer.ist.psu.edu/635250.html*, 2004.

[8] O. Othman and D. Schmidt. Optimizing distributed system performance via adaptive middleware load balancing. In *Ossama Othman and Douglas C. Schmidt, Optimizing Distributed system Performance via Adaptive Middleware Load Balancing, ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001), Snowbird, Utah, June 18, 2001.*, 2001.

[9] Ossama Othman and Douglas C. Schmidt. Issues in the design of adaptive middleware load balancing. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 205–213, New York, NY, USA, 2001. ACM Press.

[10] S. Stoller. Leader election in distributed systems with crash failures. In *S. Stoller. Leader election in distributed systems with crash failures. Technical report, Indiana University, april 1997. 169*, 1997.

[11] S. van der Zwaan and C. Marques. Ant colony optimisation for job shop scheduling. In *S. van der Zwaan, and C. Marques. Ant Colony Optimisation for Job Shop Scheduling. Proceedings of the Third Workshop on Genetic Algorithms and Artificial Life (GAAL 99), 1999.*, 1999.

AUTOMATIC DATA PATH GENERATION FROM C CODE FOR CUSTOM PROCESSORS

Jelena Trajkovic and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine
jelenat@cecs.uci.edu, gajski@cecs.uci.edu

Abstract: The stringent performance constraints and short time to market of modern digital systems require automatic methods for design of high performance application-specific architectures. This paper presents a novel algorithm for automatic generation of custom pipelined data path for a given application from its C code. The data path optimization targets both resource utilization and performance. The input to this architecture generator includes application C code, operation execution frequencies obtained by the profile run and a component library consisting of functional units, busses, multiplexers etc. The output is data path specified as a net-list of resource instances and their connections. The algorithm starts with an architecture that supports maximum parallelism for implementation of the input C code and iteratively refines it until an efficient resource utilization is obtained while maintaining the performance constraint. This paper also presents an algorithm to choose the priority of application basic blocks for optimization. Our experimental results show that automatically generated data paths satisfy given performance criteria and can be obtained in a matter of minutes leading to significant productivity gains.

Keywords: Architecture, Data Path, Design, Synthesis, C-to-RTL, Pipeline, Performance, Utilization

1. INTRODUCTION

Performance requirements for modern applications have fueled a need for specialized processors for different application domains. The reference C code typically serves as a starting point for most designs. To meet design deadlines, automatic generation of design from reference C code is needed. For most modern applications such C references are typically in the order of thousands of lines of code which is beyond the capacity of existing C-to-RTL tools. Manual hardware design is expensive and error prone process. Even though the designs are tuned to satisfy stringent performance, area and power constraints,

Trajkovic, J., Gajski, D, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 107–120.

reuse and feature extension are very difficult. On the other hand, the general purpose embedded processors are often too slow and power hungry, but offer programmability. Therefore, in this paper, we propose the automatic generation of the data path architecture based on the profile of the application and the system performance and utilization constraints. We follow a design technique for custom processors that separates the allocation of architectural resources and their connections from the scheduling of control words that drive that data path. Fig. 1 shows proposed design approach. The application code is first scheduled in an As Late As Possible (ALAP) fashion. After an application's requirements have been derived from ALAP-like schedule, they are used for allocation of the data path components. The resulting architecture is evaluated and refined using the Architecture Wizard. The Architecture Wizard iterates through the possible configurations until the given performance constraint and component utilization are satisfied.

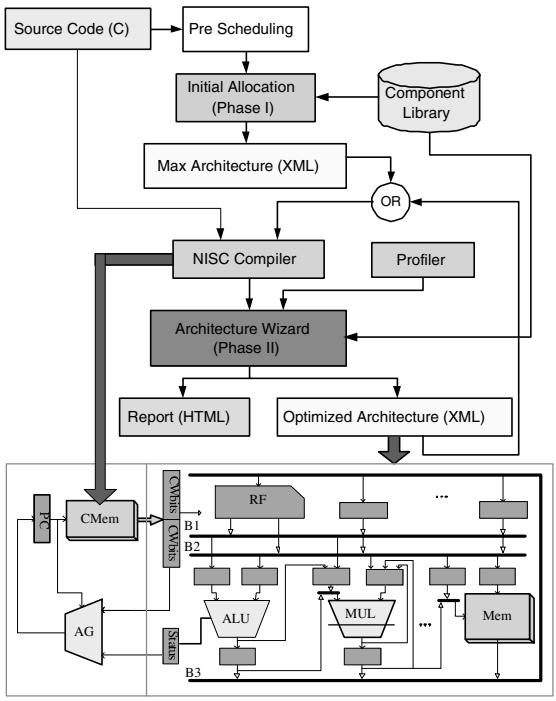


Figure 1. Custom Processor Design Technique.

2. RELATED WORK

In order to accomplish performance and power goals, ASIP and IS extension use configurable and extensible processors. One such processor is Xtensa [Tensilica: Xtensa LX, 2005], that allows the designer to configure fea-

tures like memories, external buses, protocols and commonly used peripherals [Automated Configurable Processor Design Flow, 2005; Diamond Standard Processor Core Family Architecture, 2006]. Xtensa also allows the designer to specify a set of instruction set extensions, hardware for which is incorporated within the processor [Goodwin and Petkov, 2003]. The extensions are formed from the existing instructions in style of VLIW, vector, fused operations or combination of those. Therefore, the customizations are possible only within bound of those combinations of existing instructions. This solution also requires the decoder modifications in order to incorporate new instructions. For example, having VLIW-style (parallel) instructions require multiple parallel decoders [Goodwin and Petkov, 2003], which not only increase hardware cost (that may affect the cycle time), but also limits the possible number of instructions that may be executed in parallel. However, in our approach, the decoding stage has been removed. Therefore, there is no increase in hardware complexity and no limitations on number and type of operations to be executed in parallel. In case where the code size exceeds the size of on-chip memory, instruction caches and compression techniques may be employed, both of them have been in scope of our current research.

The IS extensions, in case of Stretch processor [Stretch. Inc.: S5000 Software-Configurable Processors, 2005], are implemented using configurable Xtensa processor and Instruction Set Extension Fabric (ISEF). The designer is responsible for, using available tools, identifying the critical portion of the code ('hot spot') and re-writing the code so the 'hot spot' is isolated into the custom instruction. The custom instruction is then implemented in ISEF. Thus, the application code needs to be modified which requires expertise and potentially more functional testing. The designer is expected to explicitly allocate the extension registers. The ISEF and the main processor have the same clock cycle, and only one custom instruction/function may be generated. In contrary, our approach allows, but does not require C code modifications and does not require the designer to manipulate the underlying hardware directly.

Many C-to-RTL algorithms, such as [B. Landwehr et al., 1994; Paulin and Knight, 1989] create data path while performing scheduling and binding. [B. Landwehr et al., 1994] uses ILP formulation with emphasis on efficient use of library components, which makes it applicable to fairly small input code. [Paulin and Knight, 1989] tries to balance distribution of operations over the allowed time in order to minimize resource requirement hence the algorithm make decisions considering only local application requirements. [Devadas and Newton, 1989] takes into account global application requirements to perform allocation and scheduling simultaneously using simulated annealing. In contrast with the previous approaches, we separate data path creation from the scheduling and/or binding i.e. controller creation. This separation allows us to potentially reuse created data path by reprogramming, have controllability

over the design process and use pre-layout information for data path architecture creation.

[Gutberlet et al., 1992; Tseng and Seiwiorek, 1986; Tsai and Hsu, 1992; Brewer and Gajski, 1990; Marwedel, 1993] separate allocation from binding and scheduling. [Gutberlet et al., 1992] uses 'hill climbing' algorithm to optimize number and type of functional unit allocated, while [Tseng and Seiwiorek, 1986] applies clique partitioning in order to minimize storage elements, units and interconnect. [Tsai and Hsu, 1992] use the schedule to determine the minimum required number of functional units, buses, register files and ROMs. Than, the interconnect of the resulting data path is optimized by exploring different binding options for data types, variables and operations. In [Brewer and Gajski, 1990] the expert system breaks down the global goals into local constraints (resource, control units, clock period) while iteratively moves toward satisfying the designer's specification. It creates and evaluates several intermediate designs using the schedule and estimated timing. However, all of before-mentioned C-to-RTL techniques use FSM-style controller. Creation and synthesis of such state machine that corresponds to thousands of lines of C code, to the best of our knowledge, is not practically possible. In contrast to this, having programmable controller, allows us to apply our technique to (for practical purposes) any size of C code, as it will be shown in Section 6.

Similarly to our approach, [Marwedel, 1993] does not have limitations on the input size, since it uses horizontally microcoded control unit. On the other hand, it requires specification in language other than C and it produces only non-pipelined designs, none of which is the restriction of the proposed technique.

3. PROPOSED APPROACH

We propose a custom processor design technique for the No-Instruction-Set Computer (NISC) [Gajski, 2003]. NISC completely removes the decoding stage and stores the control words in the program memory. The NISC compiler ([Reshadi and Gajski, 2005], [Reshadi et al., 2005]) compiles the application directly onto a given data path, creating a set of control signals (called control word) that drives the components at runtime. By not having the instruction set, the data path can be easily changed, parameterized and reconfigured. Hence, the NISC concept allows separation of scheduling and allocation.

The tool flow that implements the proposed methodology is described in Fig. 1. It consists of 2 phases: the first one is performed by *Initial Allocation* and the second is implemented by the *Architecture Wizard* (*AW*) tool. In the Initial Allocation phase (Section 4), we use the schedule information to analyze component and connection requirements, and the available parallelism of a given application. The component and connection requirements are then taken

into account while choosing the instances of the available components from *Component Library* (*CL*) that will implement the data path. Resulting architecture is called Max Architecture. The Max Architecture and the application source code are used by the NISC compiler to produce the *new schedule*. The new schedule, results of the profile run and the component library are fed to the Architecture Wizard that performs estimation and refinement (Section 5). The Architecture Wizard evaluates component utilization, and uses it together with given performance and utilization constraints, to refine the existing data path architecture. The Architecture Wizard also estimates the potential performance overhead and utilization for the 'refined' architecture and automatically updates the new architecture if constraints are not satisfied. It outputs the netlist of the optimized architecture and the report in the human readable format.

4. INITIAL ALLOCATION

We start by defining maximal requirements of a given application from the application's ALAP schedule (produced by Pre-Scheduler). We choose ALAP because it gives good notion of the operations that may be executed in parallel. In addition to application's schedule, component library is another input of the Initial Allocation. The Allocator traverses the given schedule, collecting the statistics for each operation. For each operation (addition, comparison, multiplication etc.) maximum and average number of its occurrences in each cycle is computed. The tool also records the number of potential data transfers for both source and destination operands.

The Component Library consists of resources, where each one is indexed by their unique name and identifier. The record for each component also contains its type, number of input or output ports and name of each port. In case of a functional unit, a hash table is used to link the functional unit type with the list of all operations it may perform.

We derived heuristics that measure how well the available storage components match the given requirements. The heuristics use required number of source and destination operands and number of output and input ports for the storage elements available in the CL [Trajkovic et al., 2006].

While allocating functional units, we choose the type of unit that alongside the given operation, performs the largest number of operations. Thus we prevent allocation of too many units and allow the Architecture Wizard to collect statistics of operations used and potentially replace the unit with the simpler one. Once the type is decided, we allocate maximum number that is computed by the Initial Allocation tool. For example, if application requires 3 additions and 4 subtractions, and the ALU is chosen, the tool will allocate 4 instances of ALU. For practical purposes we do not allow the number of allocated units of each type to exceed the number of source buses.

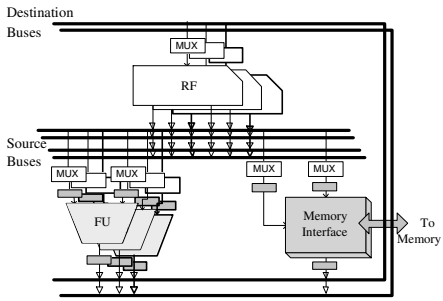


Figure 2. Components and connections.

To ensure that the interconnect is not a bottleneck in the Max Configuration, we perform a greedy allocation of connection resources (Fig. 2). This means that output ports of all register files are connected to all source buses. Similarly, input ports of all register files are connected to all destination busses. The same connection scheme applies to the functional units and the memory interface.

5. ESTIMATION AND REFINEMENT

The Architecture Wizard (AW) attempts to reduce the number of used resources to create the final design that meets given performance and utilization goals. The source code is first compiled using the Max Architecture. The resulting schedule which also has the binding information together with the execution frequencies of each basic block from the profile run is used by the AW. The following algorithm shows main steps that the AW implements.

Extract Critical Path
Create Histogram
for all Basic Block and Component Type
Label
Flatten Histogram
Estimate Overhead and Utilization
if (Overhead >= Desired Overhead and Utilization <= Desired Utilization)
Update Number of Instances of Component
Goto **Label**
Allocate Components and Create Net-list

We start by selecting the basic blocks that we want to optimize. Next, we create the histogram for each functional unit type or for each group of input or output ports of the storage unit. It is necessary to consider the utilization of all components (functional units, storage components and buses) of the same type in order to apply 'Spill' (flattening) algorithm described in the Section 5.3. For the selected blocks, we estimate the number of instances of the chosen component that will keep the execution within a given boundaries and utilization. This is repeated until both the performance and utilization constraints are met. Once the optimal number of components is decided, the output net-list is created. The following sections describe the main steps of the AW in more details.

5.1 Selection of Basic Blocks for Optimization

The goal of this phase of the Architecture Wizard is to select the basic blocks in the source code that contribute the most to the execution time and have the largest potential for optimization. The question is how to decide which basic blocks are the most promising. Our selection criteria is based on the relative size and the relative execution frequencies of the basic blocks in the application. It is likely that very large basic blocks have high potential for optimization, since they have several operations that may potentially be performed in parallel. On the other hand, even minor optimization of basic blocks that have high frequency will yield high overall performance gains. Finally, we have a class of basic blocks that have average length and average frequency, so an average reduction in their length will yield overall performance improvement that is comparable to the improvement from previous two types of optimization.

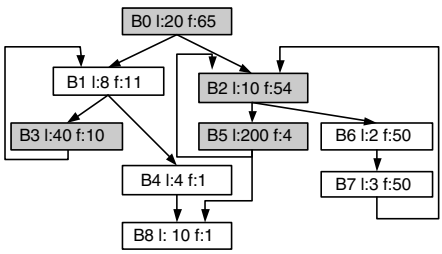


Figure 3. Selection of basic blocks for optimization.

For the Max Architecture, we use a profiler to record the execution frequency of each basic block and we use the schedule (histogram) to find the basic block's length. Following our optimization policy, we keep 3 lists of pointers to the basic blocks. The first list is sorted by frequency, the second by length and the third by frequency-length product. We use the parameterizable metrics to decide if the block is to be included in the list of blocks for optimizations. For frequency-length product we use:

$$mfl_i(x) = f_i \cdot l_i \tag{1}$$

$$M_{fl}(x) = P_{fl} \cdot \sum_{i=0}^{N} mfl_i \tag{2}$$

$$mfl_i(x) \geq M_{fl} \tag{3}$$

where f_i and l_i are frequency and length (number of cycles) of the basic block i, mfl_i is frequency-length product, P_{fl} is parameter specified by the designer and N is the total number of block in the application. The block is considered for optimizing if inequality 3 is satisfied.

In case of the list sorted by length, we observe the length of the block l_i and

$$M_l(x) = P_l \cdot \max_{i=0}^{N}(l_i) \tag{4}$$

$$l_i(x) \geq M_l \tag{5}$$

where P_l is length parameter specified by the designer. The current block is considered for optimizing if inequality 5 is satisfied.

Given f_i as the frequency of the basic block i,

$$M_f(x) = P_f \cdot \max_{i=0}^{N}(f_i) \tag{6}$$

$$f_i(x) \geq M_f \tag{7}$$

where P_f is frequency parameter. Here also, we include the block in the optimization candidate list if inequality 7 is satisfied. We supply the selected basic blocks to the Histogram Creation step of the AW.

5.2 Histogram Creation

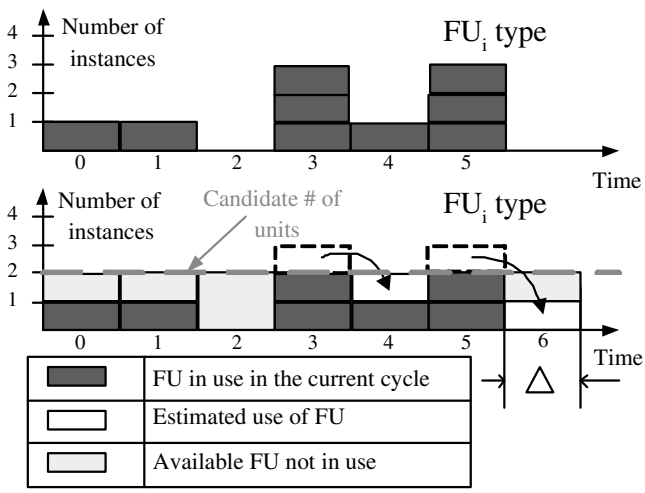


Figure 4. Example of 'Spill' Algorithm.

In this step we create a utilization histogram for each selected basic block for each component type, in case of functional units, and for data ports of the same kind (input or output), in case of the storage units. It is important to group the items of the same kind together in order to easily estimate potential execution and utilization impact when changing the number of instances. The utilization graph is extracted from the schedule generated for the Max Architecture. The example of utilization graph for the functional unit of type i is shown on the top of Fig. 4. The basic block for which the diagram is shown has 6 cycles (0 to 5). It can be seen that no instance of functional unit is used in cycle 2, one instance is used during cycles 0, 1 and 4, and 3 instances are used in the cycles 3 and 5. If we assume that the type and number of instances of all other

resources (memories, register files and its ports, buses and multiplexers) do not change, we can conclude that we need 3 instances of functional unit of type i to execute this basic block in no more than 6 cycles.

5.3 Flattening 'Spill' Algorithm

Let us assume that it is acceptable to trade off certain percentage of the execution time in order to reduce number of components (and therefore reduce area and power and increase component utilization). The designer decides the performance boundary and the desired component utilization and supplies them to the AW. The AW sets the initial value of a candidate number of components to be the largest average number of used instances of a given type for all basic blocks. The goal is to compute how many extra cycles would be required compared to the schedule with Max Architecture and what would their utilization be, if we allocate the candidate number of units. The following algorithm estimates cycle overhead and component utilization.

```
Spill (Histogram, CandidateNumber):
  for all X ∈ cycle
   CycleBudget = CandidateNumber-X.InUse;
   if CycleBudget ≥ 0
     if RunningDemand ≥ 0
      CanFit   =   MIN(CycleBudget, ABS(RunningDemand));
      RunningDemand += CanFit;
     else
      RunninBudget += CycleBudget;
    RunningDemand += CycleBudget;
```

In order to compute execution overhead and utilization we keep two counters: running demand and running budget. Running demand is a counter of operations that are scheduled for the execution in the current cycle on a unit of type i but could not possibly be executed (in the current cycle) with the candidate number of units. For example, in both cycles 3 and 5 in bottom of the Fig. 4 there is one operation that needs to be accounted for by the running demand counter (shown in dashed lines). Running budget counter counts the units that are unused in a particular cycle. In each cycle, we compare the current number of instances with the candidate number. If the current number is greater, the number of 'extra' instances is added to running demand, counting the number of operations that would need to be executed later. On the other hand, if the current number is less then the candidate, we try to accommodate as many operations as possible that were previously accounted for with the running demand counter, modeling the delayed execution. We try to fit in as many operations as possible (represented with 'CanFit' variable in the algo-

rithm) in the current cycle, as shown in the cycles 4 and 6. If there are some unused units left (when the available number of instances is greater than the running demand, like in cycles 0, 2 and 6), the running budget is updated by the number of free units.

We must note that this method does not account for interference while changing the number of instances or ports of other components. The accuracy of a given method will be discussed in Section 6. The presented estimation algorithm uses only statically available information and provides the overhead and utilization for a single execution of a given basic block. In order to be able to compare the resulting performance with the designer's requirements, we incorporate execution frequencies in the estimation.

5.4 Overhead and Utilization Estimation

```
for all C ∈ component
 while C.overhead ≤ Tspec and C.budget ≥ Uspec
  for all C ∈ component
   Spill(Histogram, CandidateNumber)
   C.overhead += B.f * RunningDemand
   C.budget += B.f * EndBudget
  Update(CandidateNumber)
```

The estimation algorithm is shown above. For each of the components, we apply the 'Spill' algorithm to all basic blocks using the largest average number of used units of a particular type across all blocks as a initial candidate number. That way we get 'per block' estimates for the overhead and utilization. Each of these statistics are multiplied by the block frequency (B.f) and accumulated in the global overhead counter (counterpart to the running demand) and global budget counter for a given unit. We also compute the dynamic length of the selected blocks for the Max Architecture by multiplying length by frequency. Having estimates for both new and the baseline architecture, we are able to decide if the candidate number of units will deliver required performance while satisfying utilization constraint. If the candidate number of units does not deliver desired performance, we increment the candidate number and repeat the estimation. If the candidate number of units is sufficient, we check the utilization, and if it is above the given threshold, we decrement the candidate number and repeat the estimation. In case the algorithm does not converge with respect to the both constraints, we give the priority to performance, and make the decision solely on the overhead.

In the simple case, shown in the the Fig. 4 if the allowed overhead is 20% (i.e. 1.2 cycles for this example) and the desired utilization per unit is 75%, having 3 units would deliver required performance, but would have the units

underutilized. Therefore, having 2 units would be satisfactory solution, with 66% utilization per unit and 17% overhead.

5.5 Allocation and Net-list creation

The allocation slightly differs from the Max Architecture allocation. The storage component allocation is done using the same heuristics with the difference that the required numbers are provided by the previous step of the AW. Previously, during the initial allocation, the operands that were appearing in the code were matched with the components from the library to determine the type of functional unit. Here the functional unit type is inherited from the Max Configuration architecture, and the number of instances is specified by the outcome of the 'Spill' algorithm.

Based on the connectivity statistics, the tool decides to provide full or limited connectivity. The full connectivity scheme is used in Max Architecture as described in Section 4. In limited connectivity scheme, we reduce number of connections from register file's output ports to the source buses, and we connect only one bus to one output port. The tool then connects the provided components according to the scheme provided in Fig. 2. It automatically allocates tri-state buffers and multiplexers as needed for the net-list that is input to NISC compiler.

6. RESULTS

We implemented the Initial Allocation and the Architecture Wizard in C++. For functional simulation of the designs, we use ModelSim SE 5.8c. The experiments were performed on a 1GHz Intel Pentium III running Windows XP. The benchmarks used are *bdist2* (from MPEG2 encoder), *Sort* (implementing bubble sort), *dct32* (from MP3 decoder) and *Mp3* (decoder). The profiling data have been obtained manually. The Architecture Wizard in presented case uses following parameters: Δ = 20%, utilization = 50%, P_{fl} = 0.5, P_f = 0.85, P_l = 0.7.

Table 1. Reduction of number of components in refined design relative to Max Architecture.

Bench.	FUs	Buses	Tri-State	Pipe. Regs
bdist2	50.0	50.0	70.0	42.9
Sort	25.0	50.0	67.7	20.0
dct32	33.3	16.7	52.5	28.6
Mp3	37.5	0.0	40.9	34.6
Avg	36.5	29.2	57.5	35.5

Table 1 gives a comparison of the number of functional units, buses, tri-state buffers, and pipeline registers of refined architecture relative to the Max Archi-

tecture for selected benchmarks. As we can see, the maximum reduction in number of functional units is 50% and the average is 36.5%. The smallest saving is for *Sort*, from 4 to 3 functional units. This is due to the limited number of components in the library: the required operations can be performed with not less than 3 units. The biggest saving is in the case of *bdist2* where half of the underutilized components are removed. The number of buses is reduced by 29.2% on an average. The is no reduction of number of buses in the case of *Mp3*. Due to the application's high parallelism, utilization of all buses if high, and therefore the algorithm does not optimize away any of them. The number of tri-state buffers is reduced by 70% in the best case and 57.5% on an average. The number of pipeline registers is reduced on an average by 35.5%. In the experiments presented here, we do not apply 'Spill' algorithm on pipeline registers, but we automatically inset them as shown in the Section 4. More sophisticated methods of deciding on how to pipeline the given architecture are topic of our research.

Table 2. Measured Δ of refined design and average number of iterations during estimation.

Bench.	Δ[%]	Avg.Iter.	T[min]	LoC
bdist2	18.8	1.8	0.9	61
Sort	0.3	3.9	0.6	33
dct32	19.6	1.2	12.6	1006
Mp3	1.1	1.4	79.5	1389
Avg	9.9	2.1	23.4	0.6

Table 2 shows the performance of resulting designs and illustrates the tool execution characteristics. The first column of the table shows the benchmark, and the next column shows measured Δ overhead in execution cycles of the refined design relative to Max Architecture. It can be seen that all benchmarks satisfy the given constraint of maximum 20% overhead. Benchmark *Sort* experiences negligible run time overhead. This application is sequential and even with the Max Architecture not more than single operation is performed in parallel. The next column shows the average number of iterations per selected basic block per component that the Architecture Wizard performs before converging. For example, on average for a given component in *bdist2* the 'Spill' algorithm will be called 1.8 times to estimate *one* selected basic block. The average number of iterations for all benchmarks is 2.1. The right most column shows the total run time required to generate both Max Architecture and refined architecture and schedule for both. The largest run time is in the order of 80 minutes, making it much faster than any hand written design.

7. CONCLUSION

We present a C-to-data path technique for designing of custom processors that alleviates the problem of manual architecture design. Our experimental results show that the generated architectures perform within overhead of 19.6% that satisfies the given performance constraint. Also, the reduction in number of used resources ranges from 29% to 58% on average across all observed components. Our algorithm performs the data path generation within 80 minutes even for large industry size application such as Mp3 decoder. The technique provides fast but effective design alternative making it possible for designers to create and evaluate many different design alternatives in less time than required for single custom design iteration. In future, we plan to add more capabilities to the automatic selection algorithm and to improve the quality of generated architectures by implementing automatic pipelining, forwarding and automatic customizing of memory hierarchy. Our current efforts are directed to overall area reduction by using multiplexer-based instead of bus-based interconnect and by minimizing the area of functional units.

Acknowledgments

The authors wish to thank Mehrdad Reshadi and Bita Gorjiara for compiler support, Verilog generator and stimulating discussions that have improved this work. We would also like to thank Pramod Chandraiah for providing the Mp3 source code.

REFERENCES

Automated Configurable Processor Design Flow (2005). Automated Configurable Processor Design Flow, White Paper, Tensilica, Inc. http://www. tensilica.com/pdf/Tools_white_paper_final-1.pdf January 2005.

B. Landwehr, P. Marwedel, and R. Dømer (1994). OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. In *Proc. European Design Automation Conference*, pages 90–95, Grenoble, France. IEEE Computer Society Press.

Brewer, F. and Gajski, D. (1990). Chippe: A system for constraint driven behavioral synthesis. *IEEE Trans. on Computer-Aided Design*.

Devadas, S. and Newton, R. (1989). Algorithms for hardware allocation in data path synthesis. *IEEE Trans. on Computer-Aided Design*.

Diamond Standard Processor Core Family Architecture (2006). Diamond Standard Processor Core Family Architecture, White Paper, Tensilica, Inc. http://www. tensilica.com/pdf/Diamond WP.pdf, October 2006.

Gajski, Daniel (October 2003). Nisc: The ultimate reconfigurable component. Technical report, Technical Report TR 03-28, University of California-Irvine.

Goodwin, David and Petkov, Darin (2003). Automatic generation of application specific processors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*.

Gutberlet, P., Müller, J., Kramer, H., and Rosenstiel, W. (1992). Automatic module allocation in high level synthesis. In *Proceedings of the Conference on European Design Automation (EURO-DAC '92)*, pages 328–333.

Marwedel, P. (1993). The MIMOLA system: Detailed description of the system software. In *Proceedings of Design Automation Conference*. ACM/IEEE.

Paulin, P.G. and Knight, J.P. (1989). Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Reshadi, M. and Gajski, D. (2005). A cycle-accurate compilation algorithm for custom pipelined datapaths. In *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

Reshadi, M., Gorjiara, B., and Gajski, D. (2005). Utilizing horizontal and vertical parallelism with no-instruction-set compiler for custom datapaths. In *In Proceedings of International Conference on Computer Design*.

Stretch. Inc.: S5000 Software-Configurable Processors (2005). Stretch. Inc.: S5000 Software-Configurable Processors http://www.stretchinc.com/products/ devices.php.

Tensilica: Xtensa LX (2005). Tensilica: Xtensa LX http://www.tensilica.com/products/xtensa_LX.htm.

Trajkovic, Jelena, Reshadi, Mehrdad, Gorjiara, Bita, and Gajski, Daniel (2006). A graph based algorithm for data path optimization in custom processors. In *Proceedings of 9th EUROMICRO Conference on Digital System Design*, pages 496–503. IEEE Computer Society.

Tsai, Fur-Shing and Hsu, Yu-Chin (1992). STAR: An automatic data path allocator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(9):1053–1064.

Tseng, C. and Seiwiorek, D.P. (1986). Automated synthesis of data paths in digital systems. *IEEE Trans. on Computer-Aided Design*.

INTERCONNECT-AWARE PIPELINE SYNTHESIS FOR ARRAY BASED RECONFIGURABLE ARCHITECTURES

Shanghua Gao[1], Kenshu Seto[2], Satoshi Komatsu[2], Masahiro Fujita[2]
[1]*Department of Electronics Engineering, University of Tokyo*
[2]*VLSI Design and Education Center, University of Tokyo*

Abstract: In this paper, we propose a novel interconnect-aware pipeline synthesis system for array based reconfigurable architectures. The proposed system includes interconnect-aware pipeline scheduling, post-placement communication scheduling and others. The experiments on a number of real-life examples demonstrate usefulness of the proposed method. For scheduling, our proposed interconnect-aware pipeline scheduling has on average only 14% overhead compared to ILP-based exact solution in terms of latency, and can achieve the same initiation interval with much less computation time. For synthesis of array based netlist with a real FPGA device, our interconnect-aware pipeline synthesis system can speed up the clock period by up to 39%, compared to a conventional high level synthesis system for array based reconfigurable architectures which utilizes loop pipelining technique but does not consider interconnect delays during scheduling phase. In addition, even when compared to a regular pipeline synthesis of general netlist, our proposed synthesis system can generate on average 18% clock period improvement.

1. INTRODUCTION

Loop pipelining is known as a very powerful technique to accelerate the execution of time-consuming loops. In deep-submicron (DSM) process era, interconnect delays (especially global interconnect delays) are becoming a more and more important factor that can affect performance, since the gate delays are getting smaller and smaller but the interconnect delays remain almost the same with continuous process scaling. Under such situation, we can not neglect this factor in high level synthesis any longer. Thus considering interconnect delays in pipeline synthesis can be a promising technique to improve the design performance. Unfortunately, to the best of the authors' knowledge, until now, few previous work considers the two factors together in high level synthesis, which may limit their improvements on the final performance.

Gao, S., Seto, K., Komatsu, S., Fujita, M, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 121–134.

There have been a number of researches in high level synthesis which exploit loop pipelining technique [1][7][8][10][11]. In recent years reconfigurable architectures have become increasingly important and are actively researched. Lee, et al [6] proposed an algorithm for mapping loops onto dynamically reconfigurable ALU arrays. Mei, et al [9] used modulo scheduling to exploit parallelism for coarse grained architectures. However, none of these methods considers interconnect delays, and they assume that all computation including interconnect delays between functional units is executed in one clock cycle. For large designs that contain long interconnect, clock period will become relatively large, which reduces their performance improvement.

Meanwhile, there are a lot of research into high level synthesis which consider interconnect delays. Xu and Kurdahi [16] proposed a method to consider layout information for FPGA based architectures by determining a set of available functional units before scheduling. They view interconnect delays as a part of one clock cycle computation. As interconnect delays become comparable or even larger than gate delays in deep submicron technology [12], Kim, et al [4][5] proposed register distributed architectures which separate interconnect delay from gate delay and allow multi-cycle interconnect delays for the first time. Cong, et al [3] improved on them and proposed a more mature architecture with high regularity called Regular Distributed Register (RDR) Architecture. They also developed corresponding architectural synthesis. Later, having the observation that for a k-cycle global interconnect, they found that it is not necessary to hold the sender register constantly for k cycles. Instead flip-flops can be inserted to the wire to relay the signal. Thus they extended the RDR architecture with pipelined interconnects by placing flip-flops on global wires. Unfortunately, none of the above work exploits loop pipelining technique, so the performance improvement for loops is limited.

We proposed a pipeline scheduling algorithm for array based architectures considering interconnect delays[13]. The algorithm is based on swing modulo scheduling (SMS) technique [8], which is a well known software pipelining technique but does not consider interconnect delays. It is a good starting point towards the research for interconnect aware pipeline synthesis. However, the work includes at least the following limitations: 1) The work evaluated the effectiveness of the scheduling algorithm only by an artificial architecture, 2) The work did not compare the proposed heuristic scheduling algorithm with an exact approach, so its optimality is unknown, 3) The work only proposed a scheduling algorithm, which is just a part of pipeline synthesis. It did not generate RTL descriptions, so the final performance is unknown. These limitations are addressed in this paper.

Our contributions are as follows:

1) We propose a novel interconnect-aware pipeline synthesis methodology to efficiently synthesize behavioral-level input into the target array-based reconfigurable architecture.

2) We compare the proposed interconnect-aware scheduling algorithm with an exact approach based on integer linear programming (ILP).

3) We present detailed evaluation of the synthesis results using a real-life FPGA device.

The rest of this paper is organized as follows. Section 2 introduces preliminaries on loop pipelining technique and target architecture. Section 3 presents our proposed interconnect-aware pipeline synthesis, including the design flow, a motivational example, the interconnect-aware pipeline scheduling and others. Section 4 shows experimental results, and Section 5 concludes the paper.

2. PRELIMINARIES

2.1 Loop pipelining technique

One main category of loop pipelining techniques is modulo scheduling [1]. The objective of modulo scheduling [11] is to generate a schedule for one iteration of a loop such that the same schedule can be repeated at regular intervals with respect to intra- and inter-iteration dependencies and resource constraints. This interval is termed *initiation interval (II)*, which reflects the performance of the scheduled loop. The inverse of the product of *II* times clock period (cp) is termed *throughput*. The larger the throughput, the faster the execution of a loop. The execution time of one iteration is termed *latency*.

Swing modulo scheduling (SMS) [7][8] is a representative modulo scheduling algorithm. It can reduce the number of registers required for the schedules. The essence of swing modulo scheduling lies in its novel ordering technique of the nodes, which enables the scheduler to place each node as close as possible to both its predecessors and successors. So the lifetimes of registers are minimized. When an operation is to be scheduled, it is scheduled in different ways depending on the neighbors of this operation in the partial schedule. Here, the partial schedule refers to the set of operations that have been scheduled previously.

2.2 Target Architecture

Figure 1 shows our target architecture, which is a two-dimensional array of islands. The size of each island is given that intra-island computation and communication can be done in a single clock cycle. In other words, the data obtained from a local register can be processed by a certain functional unit, and then be stored to a local register within one clock cycle. Inter-island data transfers can take multiple cycles.

Each island contains the following components: (1) Functional units, such as adders, multiplexers, multipliers, etc; (2) Local registers, which form the local storage elements in each island; (3) Communication interface, which carries out inter-island data transfers on a cycle-by-cycle basis; (4) Finite state machine (FSM), which provides control signals for functional units and communication interface.

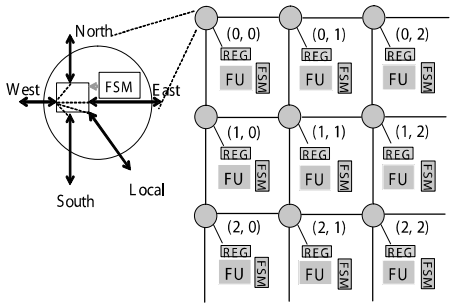


Figure 1. Target architecture

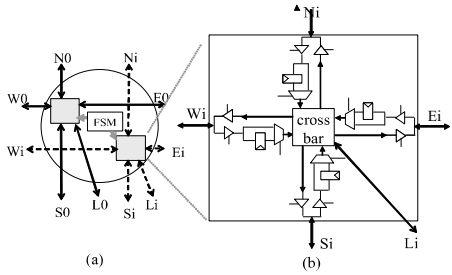


Figure 2. Communication interface

The communication interface may have several ports in each direction. We constrain that a port in one direction (eg. North) can be connected to only one port in another direction (eg. South), as illustrated by Fig. 2(a). The incoming signals to the communication interface are either relayed through pipeline registers or directly switched to different directions. For each set of connectable ports, the detailed construct is given in Fig. 2(b), which contains the following components for dynamic switching:

1) Bidirectional ports. To avoid conflict, we use tri-state buffers to clarify at which cycle a port is used as input and at which cycle it is used as output.

2) Pipeline registers. A register is allocated to each port except the *local* one, which forms the storage element for inter-island data transfers.

3) Control signals. We need control signals (provided by an FSM) for the cross bar, multiplexers and tri-state buffers.

4) Cross bar. The control signals configure the cross bar on a cycle-by-cycle basis.

The high regularity of such array based architectures simplifies the estimation of interconnect delays, which can be obtained by a function of the positions of related islands. In particular, we use the following formula to roughly estimate the delays (in terms of clock cycles):

$$w = \lceil (|x_1 - x_2| + |y_1 - y_2|)/a \rceil \tag{1}$$

Here, (x_1, y_1), (x_2, y_2) are the coordinates of the islands, and a is a parameter which represents how far (or how many islands) the signal can propagate in

one clock cycle. So, the interconnect delays are assumed to be proportional to the Manhattan distance among the islands.

3. INTERCONNECT-AWARE PIPELINE SYNTHESIS FOR ARRAY BASED ARCHITECTURES (IAPS)

In this section, we will present our proposed pipeline synthesis system, which is built on top of the target architecture. We will first introduce the overall design flow, then illustrate the benefit of considering interconnect delays in pipeline synthesis through a motivational example. Finally we will describe the key parts of this synthesis system, the interconnect-aware pipeline scheduling & placement and post-placement communication scheduling.

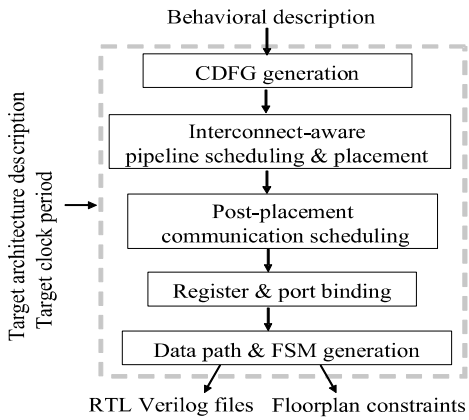


Figure 3. Overall design flow

3.1 Overall Design Flow

Figure 3 shows the overall design flow of our IAPS synthesis. The inputs are the following: 1) A behavioral description like C source code; 2) A target architecture description. This architecture description includes the dimensions of the architecture, the number and types of the FUs, the number of registers, the number of wire resources in each segment, and their location information; 3) Design constraint such as a target clock period.

At the front-end, IAPS first detects the loops from the behavioral description and generates CDFG through the intermediate representations of the low level virtual machine (LLVM) compiler infrastructure. Next, IAPS performs simultaneous pipeline scheduling & placement. To this point, we get scheduled and bound CDFG. Then, IAPS does post-placement communication scheduling.

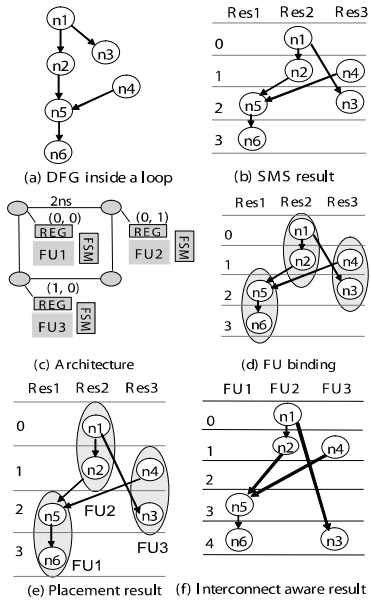


Figure 4. Motivational example

The communication scheduling is to map inter-island data transfers onto physical routing wires based on a fixed schedule and placement.

At the backend, all of the scheduling and binding information is back-annotated to the CDFG. Finally, IAPS generates data path and distributed controllers. The outputs of IAPS are: 1) A data path in a structural Verilog format and distributed controllers in behavioral FSM style. These files will be fed into logic synthesis tools. 2) Floorplan constraints. This is for the downstream place-and-route tools.

3.2 Motivational Example

In this subsection, we will illustrate the benefit of considering interconnect delays during scheduling phase of pipeline synthesis.

Figure 4(a) shows a data flow graph inside a loop. For simplicity, we assume that all nodes are of the same type (eg. addition) and are with a uniform delay (=2ns). Here we allocate 3 functional units (Res1, Res2 and Res3) with latency of 2ns. Please note that although Fig. 4(a) is a data flow graph, we can deal with a CDFG in a similar way with predicated executions [2]. According to SMS, the order for scheduling these nodes is: $< n6, n5, n2, n1, n4, n3 >$, the minimum initiation interval MII is 2, and the scheduling result is shown as

Fig. 4(b). We can see that the 6 nodes are scheduled into 4 control steps with the clock period of 2ns if interconnect delays can be ignored.

In the conventional pipeline synthesis, interconnect delays are assumed to be negligible compared to the functional unit delays. However, the assumption is no longer realistic in deep sub-micron era. Assume the architecture given as Fig. 4(c), the delay for each data link is 2ns. Thus the interconnect delay between FU2 and FU3 is 4ns, and for others it is 2ns. Recall that in the conventional synthesis flow, interconnect delays are considered as a part of clock cycle, and all data transfers complete within one clock cycle. Thus to keep the effectiveness of pipeline scheduling result, the clock period has to be increased. The naive way is to let it be the summation of the computation delay and the largest interconnect delay. For example, in this case, the largest interconnect delay is 4ns, and so increase the clock period from 2ns to 6ns.

Here, for fair comparison, we perform optimized placement after SMS so as to decrease the influence of interconnect delays on *latency* and *II*. The optimal placement consists of two steps: binding and SA-based placement. The first step is to bind operations to components such that data transfers among components with the same type are minimized as much as possible. And the second step is to map components to functional units in the given target architecture. For this example, when $n6$ and $n5$ are bound to FU1, $n1$ and $n2$ are bound to FU2 and the remaining two nodes are bound to FU3, the latency is minimum (Figure 4(d) and Fig. 4(e) show the binding result and placement result respectively). Between dependent operations $n1$ (bound to FU2) and $n3$ (bound to FU3), the interconnect delay can use control step 1, thus the clock period needs to be only 4ns. In such way, the latency becomes $4 * 4 = 16$ns, and the throughput becomes $1/(2 * 4) = 1/8$ per nanosecond.

If we consider the effect of interconnect delays during scheduling and separate interconnect delays from gate delays, we can get better results, as shown in Fig. 4(f). The *II* is also 2 but the clock period remains 2ns. Thus the latency is $5 * 2 = 10$ns, and the throughput is $1/(2 * 2) = 1/4$ per nanosecond, twice as that of the conventional flow.

3.3 Interconnect-aware pipeline scheduling and placement

Although there are a number of algorithms for loop pipelining [1], we select swing modulo scheduling as the basis to start our work, since it is able to place an operation as close as possible to both its predecessors and successors, which effectively reduce the routing length between operations. The interconnect-aware pipeline scheduling and placement algorithm is the same as the one in [13]. Please note that although the algorithm is a heuristic one, we will show that it can produce almost as good result as an ILP-based approach later.

The algorithm consists of two steps: ordering and scheduling. At present we take the same ordering technique as SMS [8]. To integrate interconnect delay effect into scheduling step, we perform scheduling and placement together. When an operation is to be scheduled, it is scheduled in different ways depending on the neighbors of this operation that are in the partial schedule. Here, the partial schedule refers to the set of operations that have been scheduled previously. For details of the algorithm, please refer to [13].

3.4 Post-placement communication scheduling

After scheduling and placement, the lifetime of data transfers and their associated sources and destinations are known. Given a set of data transfers, the number of routing resources (such as wires) and their locations, communication scheduling maps data transfers to physical wires based on a fixed schedule and placement. During this process, we need to use a modulo reservation table [11] for wires to keep track of their usage state, just as done for functional units.

As described in Section 2.2, we constrain that a port of communication interface in one direction can be connected to only one port in another direction. Given this constraint, we perform the following operations for communication scheduling:

1. Having the observation that the larger length the lifetime, the possibly more flexibility the data transfer to be scheduled, we order the data transfers in a non-decreasing of their lifetime lengths.

2. For each data transfer:

(a) Perform Maze Routing algorithm [14] to list all possible paths from source island to destination island. We take the communication interface as vertices in the grid graph [14], and assume that no vertex is blocked. The capacity of each edge is equal to the number of wires in one segment. In addition, since the estimated interconnect delays are proportional to Manhattan distance between two islands, we need only to search vertices toward the target in the exploration phase.

(b) For each possible routing path, check whether there are wires or not, to which the data transfer (from operation u to v) can be bound at consecutive $w_{(fu_{(u)},fu_{(v)})}$ control steps (equal to corresponding interconnect delay). At the same time the data transfer should start and finish within the interval $[t_{(u)}+1, t_{(v)}-1]$. Here, $fu_{(u)}$ refers to the FU that operation u is bound to, $w_{(fu_{(u)},fu_{(v)})}$ refers to the interconnect delay between FUs $fu_{(u)}$ and $fu_{(v)}$, and $t_{(u)}$ refers to the control step that operation u is scheduled to. If there are existing wires available for some path, return success, and proceed with next data transfer; Else if for every possible path, no wires available, return false, increment the number of wires in each segment, and repeat step (b).

		ILP			Proposed		
	Op	II	L(c)	rt(s)	II	L(c)	rt(s)
fir	14	2	8	0.02	2	8	0.15
filter	17	3	10	0.06	3	12	0.14
iir	24	3	11	234	3	12	0.15
wavelet	30	4	18	39	4	18	0.24
ellip	42	6	12	894	6	15	0.25
image	74	10	22	12	10	28	0.61
jfdctfst	47	-	-	-	6	16	0.28
Ave ratio	-	1	1	-	1	1.14	-

Table 1. Comparison of scheduling algorithm with ILP

3.5 Datapath & FSM Generation

After getting all the scheduling and binding information, our IAPS synthesis flow will generate data path and distributed control signals. The data path, including instances of functional units, registers, communication interface and steering logics, is in a structural verilog file. This step also generates floorplan information, which is used to constrain placement information for every instance in the data path.

In each island, an FSM controller is generated to control the instances (including the communication interface) in the island. These distrubuted controllers of different islands have identical state transition diagrams, but different output signals. The verilog files for the data path, the controllers, and floorplan constraint, are fed into logic synthesis and physical design tools to produce final layout.

4. EXPERIMENTAL RESULTS

In this section, we will first evaluate our proposed pipeline scheduling algorithm by comparing with ILP, which can generate exact results. Then we will assess our interconnect-aware synthesis system for array based architectures by comparing with a conventional high level synthesis approach which utilizes loop pipelining technique but doesn't consider interconnect delays in scheduling stage.

4.1 Proposed scheduling algorithm versus ILP

To evaluate the performance of our proposed interconnect-aware pipeline scheduling algorithm, we compared it with exact results generated by ILP (Due to page limitation, we omit the details of the ILP formulation). We tested with a 2x2 architecture, and supposed that each island contains 2 functional units.

So there are 8 functional units in total. We also assumed that the parameter a of Eq. (1) is equal to one. The test bench consists of seven programs, among which five are filter programs and two are transforms.

The results are shown in Table 1. The second column refers to the number of operations for each application, *II* represents the realized initiation interval, $L_{(c)}$ refers to the latency of one iteration in cycle, and $rt_{(s)}$ is the CPU time (in second) to compute the schedule on an Intel Xeon 3.20GHz PC. The last row is the average ratio of Proposed method over ILP. From the results we see that: (1) our proposed interconnect-aware pipeline scheduling can realize the same II as ILP, and has only 14% overhead compared to ILP in terms of latency on average. (2) Our algorithm can solve the scheduling problem in less than one second, much faster than ILP. Please note that "$-$" represents that the corresponding application kernel could not be solved by ILP within three hours.

To further show the effectiveness of our algorithm, we also tested with a larger 4x4 architecture, where each island contains 2 functional units with parameter a equal to one. We found that except the former two small examples (fir, filter), none of the applications can be solved by ILP within three hours, but our algorithm can solve the problem within several seconds. That is the reason why we use our proposed heuristic algorithm in the following experiment.

4.2 IAPS versus conventional pipeline synthesis flow

Experiment Setup. We implemented the IAPS system in C++/Linux environment. For comparison, we set up two alternative flows, as Fig. 5 shows. The left branch is a conventional pipeline synthesis for array based architectures which does not consider interconnect delays during scheduling phase. The right one is our IAPS flow discussed in this paper. In conventional synthesis flow, communication scheduling is basically the same as that for IAPS flow, except that we require that the data transfers should finish within one clock cycle.

To obtain the final performance results, *Xilinx's ISE version 6.3* [15] was used to implement the data path and controllers into a real FPGA device Spartan-3 XC3S2000. All the multipliers were implemented into the dedicated MULT blocks of the Spartan-3 device. We set the target clock frequency at *66.67MHz*, and used *Floorplanner* to constrain every instance into its corresponding island. For the compilation options, we used the default set except setting "P&R level" high.

Accounting for the regularity of the target device which contains two dedicated MULT blocks, we applied a 3x4 architecture in the experiment. Within the architecture, in column one and four each island contains 2 multipliers, and in column two and three each island contains 6 ALUs. Thus there are 12

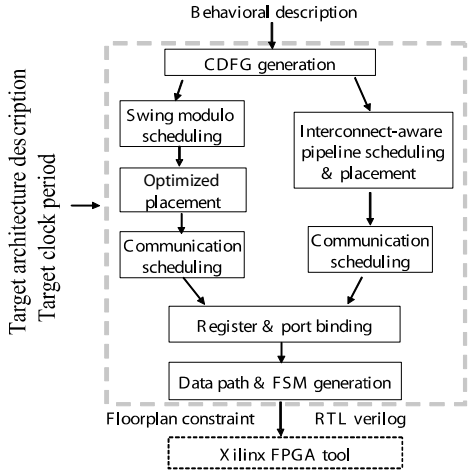


Figure 5. Two experimental flows

	Conventional			IAPS		
	Alu	Mul	Reg	Alu	Mul	Reg
iir	18	6	28	18	6	44
wavelet	27	3	48	27	3	67
ellip	21	2	44	22	2	50
jfdctfst	20	5	46	20	5	55
image	36	5	50	35	5	63
wavelet_2	23	3	36	23	4	47

Table 2. Functional unit & register usage comparison

multipliers and 36 ALUs totally. As for the benchmarks, we used the latter five applications of previous experiment, and *wavelet_2* which is obtained by unrolling the *wavelet* loop by two times.

The appropriate parameter a in Eq. (1) varies from case to case. It requires a comprehensive consideration of process technology, the architecture and base island size, and other factors. In this experiment using the Spartan-3 FPGA device with 3x4 architecture, we empirically found that it is proper to select the a as five by experiment. In future, with process technology scaling, the ratio of global interconnect delay over gate delay will become larger, and it will need multiple cycles for signals to cross the chip, as pointed out by Cong [3]. In addition, please note that Equation (1) is only one possible method to estimate the interconnect delays. In fact, our algorithm is applicable to all architectures provided that the interconnect delays can be estimated in advance.

			Conventional				IAPS			
	Op	II	cp(ns)	L(c)	Et(ns)	slices	cp(ns)	L(c)	Et(ns)	slices
iir	24	1	18.49 (1)	9	2514	2827	16.83 (0.91)	13	2356	3482
wavelet	30	1	18.65 (1)	16	2666	3919	15.56 (0.83)	23	2334	4579
ellip	42	2	21.71 (1)	10	5731	4064	16.40 (0.76)	15	4411	3953
jfdctfst	47	2	22.27 (1)	8	5834	4455	17.52 (0.79)	12	4660	4625
image	74	2	28.16 (1)	21	7744	6243	17.14 (0.61)	29	4850	5861
wavelet_2	51	2	23.07 (1)	17	6251	3762	17.46 (0.76)	26	4888	3883
Ave ratio	-	-	1	-	1	1	0.775	-	0.798	1.064

Table 3. Comparison of IAPS with a conventional pipeline synthesis flow

Mapping Results. The number of functional units and registers used during the scheduling phase is shown in Table 2. From the result, we see that the usage of multipliers and ALUs for the two flows are almost the same, but IAPS used more registers than conventional flow. This is because IAPS constrains that a FU can only access local registers of same island, yet conventional flow allows a FU to access registers located in destination island.

Results after mapping to FPGA are given in Table 3. In this table, cp refers to clock period (in ns) reported by Xilinx tool. Et refers to the execution time of a loop and $slices$ denotes the number of slices used by the FPGA implementation. And the last row refers to the average ratio of IAPS over conventional flow. Let N represent the number of iterations of a loop. The total execution time (Et) of one loop can be calculated as follows:

$$Et = (II * (N - 1) + L) * cp; \tag{2}$$

Here, we assume that each loop has 128 iterations. The results show that our IAPS flow can achieve up to 39% clock period reduction compared to conventional flow, and 22.5% reduction on average. Although the latency for one iteration may be worse than that of conventional flow, our throughput is higher so the execution time for the whole loop is improved, on average 20.2%, at the cost of some area overhead (on average 6.4% in terms of slices).

In addition, we did one more experiment to compare our proposed pipeline synthesis with regular pipeline synthesis of general netlist which does not consider interconnect delays. The results, shown in Table 4, reveal that our pipeline synthesis can improve the clock period by 18% on average, at the cost of 34% more slices used. Please note that in $Regular$ synthesis flow, we did not set any constraint on the floorplan or placement, so all its final performances are determined by ISE tool automatically.

	Regular			IAPS		
	cp(ns)	L(c)	slices	cp(ns)	L(c)	slices
iir	21.1	9	2711	16.8	13	3482
wavelet	17.6	16	3972	15.5	23	4579
ellip	19.3	10	2936	16.4	15	3953
jfdctfst	23.4	8	2859	17.5	12	4625
image	20.5	21	4402	17.1	29	5861
Ave ratio	1	-	1	0.82	-	1.34

Table 4. Comparison of IAPS with a regular pipeline synthesis of general netlist

5. CONCLUSION

In this paper, we presented a novel high level synthesis which not only exploits loop pipelining technique but also considers interconnect delays. Among this synthesis system, a key step is interconnect-aware pipeline scheduling, where we considered interconnect delays by performing pipeline scheduling and placement simultaneously. Experimentation on a number of real-life examples demonstrated the effectiveness of our interconnect-aware pipeline scheduling algorithm and our synthesis flow. For scheduling, our proposed pipeline scheduling algorithm had on average only 14% overhead compared to ILP-based exact solution in terms of latency, and could achieve the same *II* with significantly less CPU time. For the whole synthesis system, our interconnect-aware pipeline synthesis system could speed up the clock period by up to 39%, compared to a conventional high level synthesis system for array based architectures which utilized loop pipelining technique but did not consider interconnect delays during scheduling phase. Furthermore, even when compared to a regular pipeline synthesis for general netlist, our pipeline synthesis could improve clock period by 18% on average.

REFERENCES

[1] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. In *ACM, Computing Surveys*, September 1995.

[2] J. R. Allen, K. Kennedy, and J. Warren. Conversion of control dependence to data dependence. In *Proc. 10th Ann. Symp. Principles of programming languages*, January 1983.

[3] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang. Architecture and synthesis for on-chip multicycle communication. *IEEE Trans. on CAD of integrated circuits and systems*, 23(4):550–564, April 2004.

[4] J. Jeon, D. Kim, D. Shin, and K. Choi. High-level synthesis under multi-cycle interconnect delay. In *Proc. ASPDAC*, pages 662–667, January 2001.

[5] D. Kim, J. Jung, S. Lee, J. Jeon, and K. Choi. Behavior-to-placed rtl synthesis with performance-driven placement. In *Proc. Computer Aided Design*, pages 320–326, November 2001.

[6] J. Lee, K. Choi, and N. D. Dutt. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In *Proc. of LCTES*, pages 183–188, June 2003.

[7] J. Losa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: a lifetime sensitive approach. In *PACT'96*, pages 80–87, October 1996.

[8] J. Losa, A. Gonzalez, E. Ayguade, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. On Comps.*, 50(3):234–249, March 2001.

[9] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on a coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. Computers and digital techniques*, pages 255–261, 2003.

[10] P. Paulin and J. Knight. Force-directed scheduling for behavioral synthesis of asics. *IEEE Trans. on CAD*, 8(6):661–679, June 1989.

[11] B. R. Rau. Iterative modulo scheduling:an algorithm for software pipelining loops. In *Proc. the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[12] Semiconductor Industry Association. International technology roadmap for semiconductors, 2003.

[13] S. Gao, K. Seto, S. Komatsu, and M. Fujita. Pipeline scheduling for array based reconfigurable architectures considering interconnect delays. In *Proc. of ICFPT*, pages 137–144, December 2005.

[14] Naveed Sherwani. *Algorithms for VLSI physical design automation*. Kluwer Academic Publishers, 1999.

[15] Xilinx Web Site. *http://www.xilinx.com*.

[16] M. Xu and F. J. Kurdahi. Layout-driven high level synthesis for fpga based architectures. In *Proc. of DATE*, pages 446–450, 1998.

AN INTERACTIVE DESIGN ENVIRONMENT FOR C-BASED HIGH-LEVEL SYNTHESIS

Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine, CA 92697
{dongwans,gerstl, deomer, gajski}@cecs.uci.edu

Abstract: Much effort in RTL design has been devoted to developing "push-button" types of tools. However, given the highly complex nature, and lack of control on RTL design, push-button types of synthesis is not accepted by most designers. Interactive design space exploration with assistance of tools and algorithms can be more effective because it provides control of all steps of synthesis.

In this paper, we propose an interactive RTL design environment, which enables designers to control design steps. In our interactive environment, the user can control the design process at every stage, observe the effects of design decisions, and manually override synthesis decisions at will. Finally, we present a set of experimental results that demonstrate the benefits of our approach. Our combination of automated tools and interactive control by the designer results in quickly generated RTL designs with better performance than fully-automatic results, comparable to fully manually optimized designs.

1. INTRODUCTION

Automating RTL synthesis is very complicated issue. It is known that the majority of synthesis tasks are NP-complete problems. Hence, the design time becomes large, or the results are suboptimal, resulting designs cannot satisfy the performance or area demands of real-world constraints.

To develop a feasible approach for RTL synthesis, we have substituted the goal of a completely automated, "push-button" synthesis system with one that allows to maximally utilize the human designer's insights. This approach is called *Interactive synthesis methodology*. In this approach, the designer can control the design process at every stage,

Shin, D., Gerstlauer, A., Dömer, R., Gajski, D.D, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 135–144.

observe the effects of design decisions, and manually override synthesis decisions at will. This is facilitated through a convenient graphical user interface (GUI).

Hardware description languages (HDLs) such as Verilog HDL and VHDL are most commonly used as input to RTL design. However, system designers often write models using programming languages such as C/C++ to estimate the system performance and to verify the functional correctness of the design, even to refine the design into implementation.

C/C++ offers fast simulation as well as a vast amount of legacy code and libraries which facilitate the task of system modeling. To implement parts of the design modeled in C/C++ in hardware using synthesis tools, designers must then manually translate these parts into a synthesizable subset of a HDL. This process is well known for being both time consuming and error prone. Moreover, it can be eliminated completely. The use of C-based languages to describe both hardware and software will accelerate the design process and facilitate the software/hardware migration. Hardware synthesis tools from C/C++ can then be used to map the C/C++ models into logic netlists.

The rest of the paper is organized as follows: section 2 shows related work and section 3 introduces our RTL design environment and the program flow of the proposed RTL synthesis tool. Section 4 shows the experimental results. Section 5 concludes the paper with a brief summary.

2. RELATED WORK

Issues in RTL modeling, RTL design and behavioral synthesis, aka. High-Level Synthesis (HLS), have been studied for more than a decade now [3].

In the recent years, a few projects have been looking at means to use C/C++ as an input to current design flows [4, 6, 16]. Constructs are added to model coarse-grain parallelism, communication and data-types. These constructs can either be defined as new syntactic constructs, hence creating a new language [4]. They can also be implemented as part of a C++ class library [6]. In order to facilitate the mapping of C/C++ models into hardware, several tools exist that automatically translate C/C++ based descriptions into HDL either at the behavioral level or the register transfer level (RTL) [13, 16, 7].

Many automatic synthesis tools (also known as *push-button* synthesis) have been developed, including Olympus [10], OSCAR [11], SPARK [7], Synopsys Behavioral Compiler [15], Mentor Catapult-C [12], and Cyber [16]. However, these tools provide no means to access the interme-

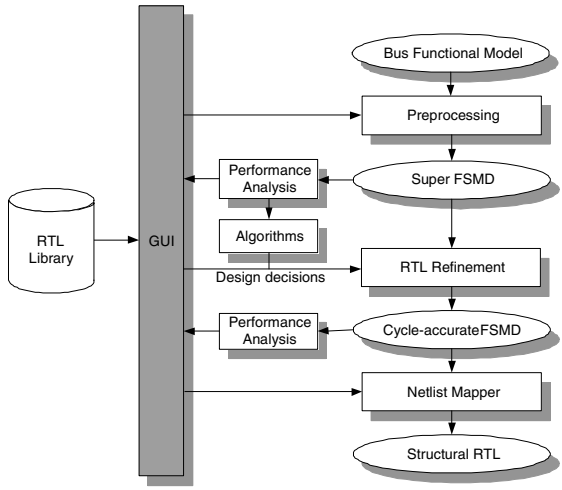


Figure 1. RTL design flow

diate design models that are created during the synthesis process and to change important decisions by designers. The designer can specify design constraints for whole designs and access the behavioral input model and the structural output model and design constraints.

Some interactive synthesis approaches [8, 9] addressed the importance of user-interaction with synthesis system. However they have a fixed design flow, that is, the designer has to perform a sequence of synthesis tasks in a predefined order and a cycle-accurate simulation model with complex components is not available for the intermediate stages.

3. RTL DESIGN ENVIRONMENT

In this section, we will describe our RTL design environment integrated in a system-level design flow. The RTL design environment provides synthesis, refinement and exploration for RTL design as shown in Figure 1. It includes a graphical user interface (GUI) and a set of tools to facilitate design flow and perform refinement steps. In our flow, designers or algorithms of automatic tools can make decisions such as clock period selection, allocation, scheduling and binding. The GUI allows designers to input and change such design decisions. It also enables the designer to observe the effects of the decisions and to manually override the decisions at will. Further, the designers can make partial decisions and then run automatic tools to take care of the rest of the decisions.

We model an RTL design as a Finite State Machine with Data (FSMD) [1], which is an FSM model with assignment statements added to each state. The FSMD can completely specify the behavior of an arbitrary

RTL design. The variables and functions in the FSMD may have different interpretations which in turn defines several different styles of RTL semantics.

In addtion, in order to represent pipelined or multicycled units in a design in the cycle-accruate FSMD, we have introduced new constructs [2] such as `piped` for pipelined units, and `after` for multicycle units. The simulation speed of the cycle-accurate FSMD is significantly improved compared with that of the structural RTL description.

During preprocessing, the behavioral description of custom hardware in C/C++ will be refined into an SFSMD model where each state is a basic block of the original description. Also some presynthesis optimization techniques including constant propagation, dead code elimination, and common subexpression elimination are integrated. The generated FSMD will be the input model of the RTL synthesis.

A performance analysis tool is used to obtain characteristics of the initial design such as the number of operations, variables and data transfers in each state, which serves as the basis for RTL design exploration. It also produces quality metrics for RTL design such as the delay and power of each state and area of the design to help the designers to make decisions on clock selection, allocation, scheduling and binding.

The refinement tool then automatically transforms the FSMD model based on relevant design decisions. Finally, the structural RTL model is produced by a netlist mapper, ready to feed into traditional design tools for logic synthesis, etc.

3.1 Synthesis Decisions

The refinement engine works on directions called the RTL synthesis decisions. The synthesis process can either be automated or interactive as per the designer's choice. However, the decisions must be input to the refinement engine using a specific format. For the purpose of our implementation, we annotated the input model with the set of synthesis decisions. The refinement tool then detects and parses these annotations to perform the requisite model transformations. Based on these decisions, the refinement engine imports the required RTL components from the RTL component library and generates the cycle-accurate FSMD.

The decisions can be made by designers interactively through GUIs and/or be made through automatic algorithms. The GUIs for interactive decision-making allows designers to (a) specify decisions (b) override the decisions, which are already made by the designers or automatic algorithms (c) partially assign decisions and automatic algorithms will fill in the rest of decisions.

The GUI also allows automatic algorithms being plugged in. Thus it is easily extendible because designers can select an algorithm from a list of plug-in algorithms such as ASAP, ALAP, list and force-directed scheduling and graph coloring for binding and so on.

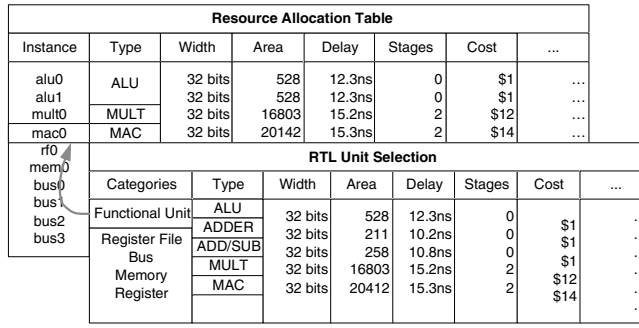


Figure 2. Allocation window

3.1.1 GUI for Interactive Decision-making. In order to help designers to make synthesis decisions interactively, we provide an *allocation window* and a *scheduling & binding window*. In allocation window as shown in Figure 2, designer can see all RTL components in the RTL component library, select them and set the parameters such as bit width, size of array and so on [5].

The scheduling & binding window displays the SFSMD in *state-operations table* format which contains a series of states, each state containing a set of operations to be performed in the state, shown in Figure 3. The state-operations table displays the behavior of a design and all design decisions made in graphical format. This is, the designer can modify all design decisions at any time in the design process in the state-operations table. In the table, *State* is the current state and *NS* is next state. *CS* is the control step of the expression which is relative to the start time of the state.

The table also shows statistics such as the lifetimes of all variables, occurrences of operations, the number of data transfers and the critical path in number of operations in each state. It also shows the ASAP and ALAP control step for each expression in each state.

All expressions are scheduled at specified control steps in the scheduling view, which will be assigned to *CS* in the state-operations table. All operations are bound to functional units and their ports, which will be specified in the *oper* column. Also all operand variables (*destination*, *source1*, *source2*) are mapped to storage units, read/write ports of the

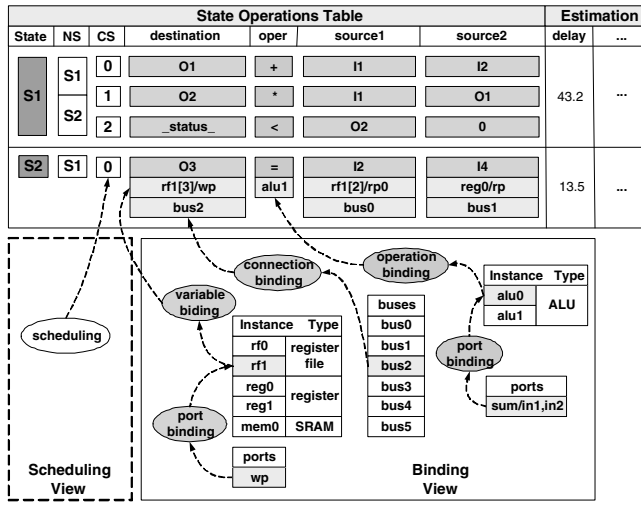


Figure 3. Scheduling & Binding window for an SFSMD

storage unit, and busses in the binding view. If the variables are mapped to memory, then the base address needs to be specified as well.

Designers can input, modify all decisions and override decisions which algorithms made through automatic tools in scheduling & binding window. Furthermore, the designers can partially specify some of the decisions and then algorithms take care of the rest of decisions still meeting the specified designer's decisions.

3.2 Performance Analysis

Several synthesis metrics are implemented to help the designer decide how to select the allocation and partition a super FSMD description into control steps. Two important metrics of design cost are operator occurrences and variable lifetimes. Operator occurrences metric shows the number of operations of each type used in each state. The maximum number of occurrences of a certain operator type over all states determines the required minimum number of functional units to perform that type of operation. Variable lifetimes metric identifies states in which state a variable holds a useful value. The maximum number of variables with overlapped lifetimes over all states determines the required minimum number of storage units. After allocation, performance estimation calculates the delay and power consumption of each state and the area of the design.

4. EXPERIMENTAL RESULTS

We have implemented our interactive RTL synthesis approach in C++ (algorithms, data structure) and Python (GUI). The benchmarks used are *sra* (square root approximation), *GCD* (greatest common divisor), *DIFFEQ* (differential equation solver), from high-level synthesis benchmark suite.

The different types of implementation of discrete cosine transformation, *DCT* (2-dimensional DCT with matrix multiplication), *ChenDCT* (2-dimensional DCT implementing Chen algorithm), *MP3DCT* (1-dimensional DCT for MP3 Codec) *MP3IMDCT* (1-dimensional ImDCT for MP3 Codec) *Codebook* (codebook search block in the GSM Vocoder which is employed worldwide for cellular phone networks. The model was based on the bit-exact reference implementation of the ETSI standard in ANSI C).

Table 1 lists the characteristics of the designs used in terms of the number of operations (`#OPs`) in the input description, which is indicative of the data complexity of the design, and the number of basic blocks (`#BBs`), indicative of its control complexity. Also the type and quantity of each resource allocated to schedule and bind this design for all the experiments are given in Table 1. The resources indicated in this table are: *ALU* is used for arithmetic and logic operations ($+$, $-$, &, $|$, $\hat{}$, and $\sim$) or saturated arithmetic operations in `Codebook` example. *ADD* and *SUB* for addition ($+$) and subtraction ($-$) respectively, *ASU* for both addition and subtraction, *MULT* for multiplication ($\times$) in 1 stage pipeline fashion, *DIV* for division ($\div$) in 5 stage pipeline fashion, *SQRT* for square root operation in 5 stage pipeline fashion, *LU* for logic operations (&, $|$, $\hat{}$, and $\sim$), *SHIFT* for left/right shift operation ($\ll$, $\gg$), *CMP* for comparison ($\geq$, $\leq$, , $<$, $==$, $!=$, !)

REG is a register with 1 read port and 1 write port and *RF* is a register file with 2 read ports and 1 write port. *MEM* and *RAM* have 1 read port and 1 write port, and the *ROM* has 1 read port. The number in parenthesis indicates the size of register files and memories.

For our experiments, we implements a heuristic based on list scheduling algorithm [14] which performs scheduling and binding at the same time (`Automatic` in Table 2). The heuristic considers the allocation of units and the number of ports of allocated units and busses. After applying the heuristic algorithm, we applied some optimization techniques to improve the performance of designs on the RTL design environment (`Automatic + Manual` in Table 2), while `Manual` shows the examples designed by designers.

For `MP3DCT` and `MP3IMDCT`, a designer manually implements control pipelining by inserting pipeline registers on the output logic of the controller, which can reduce the clock period by reducing the delay from the output of output logic of the controller to its datapath. In addition, all control words are stored in program ROM.

For `Codebook`, the designer inserts registers at the output of functional units to reduce the clock period and implements data forwarding from output of a function unit to the input of other functions units. In addition, the special counters are introduced to calculate the index of arrays in the memory (inside loops), which is accessed by row and column by two indices.

We present the logic synthesis result obtained after synthesizing the RTL Verilog generated by Netlist mapper using the Synopsys *Design Compiler* logic synthesis tool. The `LSI-10K` synthesis library is used for technology mapping and components are allocated from the Synopsys DesignWare Foundation library.

The logic synthesis results are presented in terms of three metrics: the unit area (in terms of synthesis library used), the number of states (`states`) in FSM controller, the critical path length (clock period, `CP` in nanoseconds) and the number of clock cycles (`cycles`) to execute the design. The critical path length is the length of the longest combinational path in the netlist as reported by static timing analysis tool and it dictates the clock period of the design.

Table 1. Characteristics of synthesis examples

Examples	#OPs	#BBs	resources
GCD	5 (3)	9	1 SUB, 1 CMP, 1 LU, 4 REGs
DIFFEQ	11 (10)	6	1 ASU, 1 CMP, 1 MULT, 16 REGs
DCT	21 (8)	22	1 ADD, 1 MULT, 3 COUNTs, 3 REGs, 3 RAM (64), 2 ROM (64)
ChenDCT	167 (42)	27	1 ADD, 1 SUB, 1 SHIFT, 1 MULT, 1 CMP, 16 REG, 1 RAM (128)
MP3DCT	330 (0)	3	1 ASU, 1 SHIFT, 1 MULT, 1 CMP, 2 RFs (64)
MP3IMDCT	195 (178)	49	1 ASU, 1 SHIFT, 1 MULT, 1 CMP, 2 RFs (64)
Codebook	604 (253)	197	1 ALU, 1 CMP, 1 MULT, 1 DIV, 1 MAC, 1 SQRT, 16 REGs, 1 RAM (2048)

The speedup, as shown in Table 3 of a design by `Automatic + Manual` and `Manual` is defined as follows:

$$Speedup_x = \frac{CP_{Auto.} \times cycles_{Auto.} - CP_x \times cycles_x}{CP_{Auto.} \times cycles_{Auto.}} \times 100$$

Table 2. Synthesis result

Examples	Automatic			Automatic + Manual			Manual		
	area	CP	cycles	area	CP	cycles	area	CP	cycles
GCD	5154	35.7	34	5177	33.9	34	—	—	—
DIFFEQ	17484	37.6	113	16314	31.7	93	—	—	—
DCT	95408	60.1	6914	95173	50.9	5792	114090	42.6	4225
ChenDCT	32487	49.0	2469	31155	45.4	2129	—	—	—
MP3DCT	85786	61.2	565	83780	54.9	308	103357	47.7	308
MP3IMDCT	93069	55.0	382	91669	50.9	209	70239	44.3	209
Codebook	991215	71.4	56492	991065	70.2	43195	987162	55.5	33000

Table 3. Performace improvement

Examples	Automatic + Manual	Manual
GCD	5.0%	—
DIFFEQ	30.6%	—
DCT	29.1%	56.7%
ChenDCT	20.1%	—
MP3DCT	51.1%	57.5%
MP3IMDCT	49.4%	55.9%
Codebook	24.8%	54.6%

where x is either `Automatic + Manual` or `Manual`.

Through the experiments, the performace of the design by the heuristic algorithm is the worst among 3 implementations, but after modified the scheduling and binding result by users, the designs get about 30% improvement, specially in terms of clock cycles, and become close to the designs by human.

5. CONCLUSION AND FUTURE WORK

In this paper, we proposed an interactive C-based RTL design environment which takes full advantage of the designer's insight by allowing to enter, modify, override all decisions at will.

It has been developed and integrated into SoC design environment in order to validate our approach. This allows designers to evaluate several design points during fast exploration. Our experimental results show that the proposed solution to improve behavioral modeling of RTL designs is not only feasible and practical for real-world designs, it also comes with a significant speed-up in simulation.

Future work in this direction will involve comparison between our approach and commercial tools and the scheduling of bus protocols under timing constraint in clock cycles.

REFERENCES

[1] Accellera C/C++ Working Group of the Architectural Language Committee. RTL Semantics, Draft Specification. Technical report, Accellera, February 2001. available at http://www.eda.org/alc-cwg/cwg-open.pdf.

[2] R. Dömer, A. Gerstlauer, and D. Shin. Cycle-accurate RTL modeling with multi-cycled and pipelined components. In *Proceedings of International SoC Design Conference*, pages 375–378, October 2006.

[3] D. D. Gajski, N. Dutt, S. Y.-L. Lin, and A. Wu. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[4] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, January 2000.

[5] A. Gerstlauer, L. Cai, D. Shin, R. Dömer, and D. D. Gajski. System-on-Chip component models. Technical Report CECS-TR-03-26, Center for Embedded Computer Systems, University of California, Irvine, August 2003.

[6] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, March 2002.

[7] S. Gupta. SPARK: High-level synthesis using parallelizing compiler techniques. available at http://www.cecs.uci.edu/~spark/.

[8] A. A. Jerraya, I.-C. Park, and K. O'Brien. AMICAL: An interactive high level synthesis environment. In *Proceedings of the European Design Automation Conference*, pages 58–62, February 1993.

[9] H.-P. Juan, D. D. Gajski, and V. Chaiyakul. Clock-driven performance optimization in interactive behavioral synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–157, November 1996.

[10] D. Ku and G. D. Micheli. *High-level Synthesis of ASICs under Timing and Synchronization Constraints.* Kluwer Academic Publishers, 1992.

[11] B. Landwehr, P. Marwedel, and R. Dömer. OSCAR: Optimum simultaneous scheduling, allocation and resource binding based on integer programming. In *Proceedings of the European Design Automation Conference*, February 1994.

[12] Catapult C Synthesis, Mentor Graphics Inc. available at http://www.mentor.com/.

[13] L. Séméria and G. D. Micheli. SpC: Synthesis of pointers in C, application of pointer analysis to the behavioral synthesis from C. In *Proceedings of the International Conference on Computer-Aided Design*, pages 340–346, November 1998.

[14] D. Shin and D. D. Gajski. Scheduling in RTL design methodology. Technical Report CECS-TR-02-11, Center for Embedded Computer Systems, University of California, Irvine, April 2002.

[15] Behavioral Compiler, Synopsys Inc. available at http://www.synopsys.com/.

[16] K. Wakabayashi and T. Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vender perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, December 2000.

INTEGRATED COUPLING AND CLOCK FREQUENCY ASSIGNMENT OF ACCELERATORS DURING HARDWARE/SOFTWARE PARTITIONING

Scott Sirowy and Frank Vahid*
Department of Computer Science and Engineering - University of California, Riverside
*Also with the Center for Embedded Computer Systems, University of California, Irvine

Abstract: Hardware/software partitioning moves software kernels from a microprocessor to custom hardware accelerators. We consider advanced implementation options for accelerators, greatly increasing the partitioning solution space. One option tightly or loosely couples each accelerator with the microprocessor. Another option assigns a clock frequency to each accelerator, with a limit on the number of distinct frequencies. We previously presented efficient optimal solutions to each of those sub-problems independently. In this paper, we introduce heuristics to solve the two sub-problems in an integrated manner. The heuristics run in just seconds for large examples, yielding 2x additional speedup versus the independent solutions, for a total average speedup 5x greater than partitioning with a single coupling and single frequency.

1. INTRODUCTION

Partitioning an application's kernels to execute on a custom hardware accelerator rather than on a microprocessor—known as hardware/software partitioning—is a well-known technique for improving application performance [1] and improving energy consumption [2]. Such partitioning is relevant to both ASIC (application-specific integrated circuit) and FPGA (field-programmable gate array) implementation. The rise of FPGAs in commercial microprocessor platforms [3] makes such partitioning increasingly important.

Sirowy, S., Vahid, F, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 145–154.

Most previous hardware/software partitioning approaches did not consider different couplings of the accelerators with the microprocessor. However, modern platforms, including FPGAs, support at least two couplings. *Tightly coupled* accelerators have direct access to the microprocessor memory or cache, and thus operate at a single clock frequency, which will necessarily be the slowest frequency of any of those accelerators. *Loosely coupled* accelerators instead access memory through a bridge, and thus may each have unique optimized clock frequencies. Thus, there exists a tradeoff to couple an accelerator tightly or loosely based on the importance of sinlge cycle memory access or running at the fastest possible clock frequency. Figure 1 shows a typical architecture that supports multiple couplings. The two tightly coupled accelerators have single cycle access to memory at the expense of both being clocked at 58 MHz even though one could have been clocked at 166 MHz. We refer to the problem of coupling a set of accelerators tightly or loosely as the *two-level microprocessor-accelerator partitioning problem.*

Modern platforms, including FPGAs, may support several different frequencies on a single chip. For example, the Xilinx Spartan 3 supports four distinct clock frequencies, while the Xilinx Virtex II supports up to eight [4]. Much current research investigates methods to take advantage of multiple clock domains for heterogeneous core architectures, systems-on-a-chip, etc., for both performance and energy benefits [5]. However, the number of accelerators often exceed the number of available clock frequencies. In this case, the accelerators must be grouped to share clock frequencies, necessarily running at the slowest frequency of the group. For example, in Figure 1, the four loosely coupled accelerators must share two clock frequencies. We refer to the problem of assigning a fixed number of clock frequencies for minimal application execution time as the *clock frequency assignment problem*.

Most previous approaches do not consider clock frequency assignment for the accelerators. While the tightly coupled accelerators should all execute using the same frequency, the loosely coupled accelerators could potentially each execute with different frequencies. In previous work, we solved the coupling assignment problem optimally, assuming enough available clock frequencies to support unique frequencies for each loosely coupled accelerator [6]. In a separate work, we solved the problem of assigning a limited number of frequencies to the set of loosely coupled accelerators such that performance was maximized [7]. In this work, we show that solving the two problems in an integrated manner can yield significant performance improvements over solving them sequentially

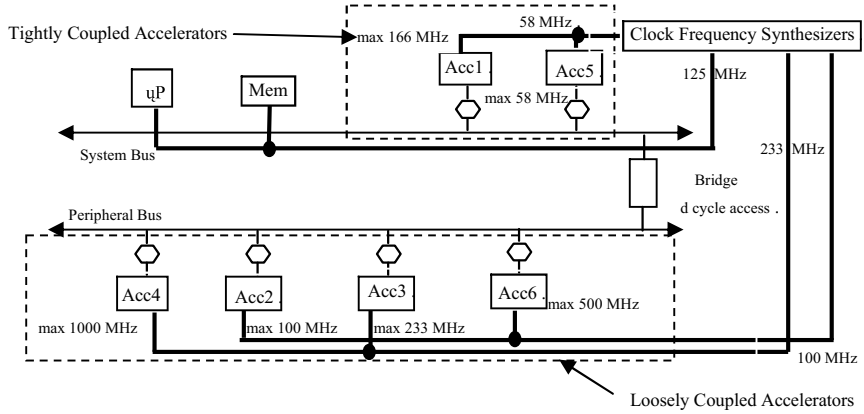


Figure 1. A two-level system architecture that is driven by four clock frequencies. The system bus has two tightly coupled accelerators that run at a slower clock frequency but have single cycle access to memory.

2. PROBLEM DEFINITION AND DESCRIPTION

We previously solved the two-level microprocessor-accelerator partitioning problem and the clock frequency assignment problem optimally using novel dynamic programming techniques for each. This section reviews those solutions, and then defines a new problem integrating both problems.

2.1 Two-Level Microprocessor-Accelerator Partitioning

The problem of partitioning accelerators to either a tightly coupled set or a loosely coupled set, assuming that each loosely coupled accelerator could run at its own unique clock frequency, used the following objective function for minimizing the execution time of all the accelerators:

$$TC([\sum_{i=1}^{n}(comp\_cycles_i + mem\_accesses_i)]/\min\_clock)$$

$$+ LC(d * \sum(mem\_accesses_i / clk\_freq_i)$$

$$+ \sum_{i=1}^{n}(comp\_cycles_i)/clk\_freq_i))$$

We solved our problem optimally using a novel dynamic programming algorithm we refer to as the *n-knapsack dynamic programming*, or ***NKDP***, solution. A complete description of the solution is given in [6].

2.2 Clock Frequency Assignment Partitioning

In the clock frequency assignment problem, we again considered a set of accelerators *A* which had already been determined by a previous hardware/software partitioning decision. Given a maximum number of unique clock frequencies *F* available to the accelerators, the *clock-frequency assignment problem* is to:

Find a positive integer value for every a_i.freq, such that each a_i.freq is less than a_i.maxfreq for every i, the number of distinct a_i.freq values is less than or equal to F, and the execution time E is minimized.

We also developed a novel dynamic programming algorithm to solve the clock partitioning problem optimally. The complete solution description is given in [7].

2.3 Integrated Two-Level Partitioning and Clock Frequency Assignment

The integrated coupling and clock frequency assignment problem takes as input a set of functions to be implemented as accelerators, determined by a previous hardware/software partitioning decision (our problem and partitioning may iterate). Each accelerator is annotated with four numbers, determined from synthesis and simulation of each function: The number of memory accesses, the total number of computation cycles, the synthesized area, and the maximum possible clock frequency. The number of memory accesses and computation cycles may represent averages or worst-case numbers, depending on whether the designer seeks to optimize for overall average or worst-case performance.

The extra cycles of the bridge is also given. This memory access penalty is an architectural feature of the bridge, and not a per-application number, so the number is fixed for all applications. A loosely coupled accelerator would incur this latency penalty each time it made an access to memory, since the accelerator is connected to the memory through the bridge.

All tightly coupled accelerators, having single-cycle access to memory or cache, must run at a single clock frequency – this assumption matches several modern commercial FPGAs that incorporate microprocessors. Because all those accelerators must run at one clock frequency, they all must run at the frequency of the *slowest* tightly coupled accelerator in the group.

The tightly coupled accelerators' frequency need not be the same as the microprocessor's frequency.

Loosely coupled accelerators, in contrast, could potentially run at their unique, fastest clock frequency. However, since modern FPGA platforms impose a limit on the number of available clock frequencies, several of the loosely coupled accelerators may also need to be merged together and share the same clock frequency. This means several of the accelerators will not be able to run at their own unique clock frequency. The number of available clock frequencies F is usually given in the documentation for the particular FPGA being used. For instance, a Xilinx Spartan 3 board supports up to four unique clock frequencies, while the Xilinx Virtex II supports up to eight clock frequencies.

Formally, the problem takes as input a set of accelerators $A=\{a_1,a_2,\ldots,a_n\}$. Each function is annotated with several different weights: $a_i.comp\_cycles$, $a_i.mem\_accesses$, $a_i.area$, $a_i.max\_freq$, and $a_i.frequency$. The term $a_i.frequency$ is not given and must be determined. The memory access penalty through a bridge is given as d, and the number of available clock frequencies is given as F. The objective function is to thus minimize the application execution time as follows:

Find a positive integer value for every $a_i.freq$, such that each $a_i.freq$ is less than $a_i.maxfreq$ for every i, the number of distinct $a_i.freq$ values is less than or equal to F, one group has single cycle access to memory while the rest have d cycle access, and the execution time E is minimized.

3. HEURISTICS

We present two heuristics to solve the clock frequency assignment problem for two-level microprocessor-accelerator platforms. Before that, a straightforward *sequential* approach performs two-level microprocessor-accelerator assignment first assuming unlimited distinct clock frequencies, followed by clock frequency assignment on the loosely coupled accelerators with (F-1) clock frequencies (since one clock frequency must necessarily be used for the tightly coupled accelerators). Each sub-problem can be solved optimally using our previous techniques.

Because the running time of NKDP is $O(Sn^2)$, where S is the area constraint, and the running time of the clock frequency assignment algorithm is $O(nF^2)$, the overall worst case time complexity of the sequential approach is $O(Sn^2 + nF^2)$. However, since the assumption that the two-level microprocessor-accelerator partitioning algorithm can operate every loosely coupled accelerator at its own distinct clock frequency is potentially

violated, the two level partitioning becomes suboptimal, and therefore the entire solution is suboptimal.

3.1 No Penalty Migration

Our first heuristic was based on the observation that when the ***NKDP*** algorithm partitions the accelerators into both a tightly coupled and loosely coupled set, there may be accelerators in the loosely coupled set that are clocked with a faster maximum frequency than the tightly coupled set. This is because the NKDP algorithm decided that having a faster frequency was more important than having single cycle access to memory. However, with the number of clocks constrained in clock frequency assignment, that accelerator's frequency may be reduced below the tightly coupled clock set frequency. Thus, migrating the accelerator from the loosely coupled set to the tightly coupled set makes sense (assuming it fits the area constraint) since the accelerator would run faster as a tightly coupled accelerator than merged with a slower accelerator in the loosely coupled set. Because the accelerator's fastest possible frequency is faster than the already established tightly coupled set clock frequency, the heuristic can migrate the accelerator to the tightly coupled set at no penalty to the tightly coupled set. We call this *No Penalty Migration*. After the heuristic migrates an accelerator from the loosely coupled set to the tightly coupled set, clock frequency assignment is again run on the remaining accelerators in the loosely coupled set to determine if a new assignment exists, since one less accelerator may result in a better partitioning of the available clock frequencies to the remaining loosely coupled accelerators.

3.2 Nested Dynamic Programming

We also developed a heuristic in which we tried to integrate the two solutions by having the two-level microprocessor-accelerator algorithm call the clock frequency assignment algorithm each time the knapsack algorithm returns a possible solution. We call this the *Nested Dynamic Programming* heuristic. The *No Penalty Migration* heuristic assumes the initial two-level microprocessor-accelerator partitioning chose the best two-level assignment, meaning the tightly coupled frequency should be maintained.

However, the clock frequency assigned to the tightly coupled accelerators may not be optimal when considering the clock frequency assignment problem too, and thus no amount of clock frequency assignment and migration on the remaining accelerators would result in the optimal solution. Because the two-level microprocessor-accelerator dynamic programming

algorithm runs knapsack *n* times, resulting in *n* potential solutions, running the clock frequency assignment dynamic programming algorithm on each of those solutions would result in a more accurate solution space, since more options are allowed into the tightly coupled accelerator set.

The solution to each knapsack is passed to the clock frequency partitioning algorithm. The clock frequency assignment algorithm determines the clock frequency assignment for the loosely coupled accelerators. The best solution is maintained and returned. We note the "best" solution is returned as opposed to the "optimal" solution from the original NKDP algorithm, because the heuristic still potentially violates the assumption that the NKDP algorithm assumes each loosely coupled accelerator can run at its own distinct clock frequency. The heuristic is only guaranteed to return optimal results when the number of clock frequencies exceeds the number of accelerators that require a distinct clock frequency. The worst case running time of the nested dynamic programming heuristic is also $O(n^2 (S + F^2))$, since the nested dynamic programming algorithms run the clock frequency assignment algorithm *n* times.

4. EXPERIMENTS AND RESULTS

This section describes results of applying the two heuristics to a commercial quality H.264 video decoder from Freescale Semiconductor. We implemented the heuristics on a 2.66 GHz 1GB RAM Pentium 4 PC. We targeted synthesis to a Xilinx IV Pro, and gathered information on cycles per function and maximum clock frequency of each accelerator. We also tested our heuristics using a wide range of synthetic benchmarks.

H.264 is a proprietary video decoder developed by the Video Coding Experts Group (VCEG), and part of the MPEG-4 standard. Unlike common benchmarks taken from publicly available reference implementations, the decoder's code was highly optimized, and thus did not consist of just two or three critical functions, but rather of 42 critical functions that together accounted for about 90% of execution time. We utilized Stitt's partitioning into accelerators [9], which was straightforward, involving implementing an accelerator for each critical function. We gathered computation cycle and memory access information through synthesis and simulation, and clocked each accelerator targeted for Xilinx's Virtex IV Pro. The variation in maximum frequencies ranged from 40 MHz to 285 MHz.

Figure 2 shows the results running the heuristics on the highly optimized H.264 video decoder. The speedups are normalized to results when all accelerators use only one clock frequency and one coupling. Figure 2 shows that one additional clock frequency allowed the heuristics to couple the 42

accelerators either tightly or loosely, and thus gain a 3.5x speedup over the single frequency, single coupled implementation. The inclusion of additional clock frequencies further improves the speedup to almost 4x. For the H.264 application, the *No Penalty Migration* and *Nested Dynamic Programming* heuristics performed similarly, attaining almost the same speedup. Although both heuristics have the same worst case runtime, the *No Penalty Migration* heuristic consistently attained results faster than the *Nested Dynamic Programming* heuristic. We also note that as the number of clock frequencies increases, the improvements of both the *No Penalty Migration* and *Nested Dynamic Programming* heuristics compared to the sequential approach become almost negligible.

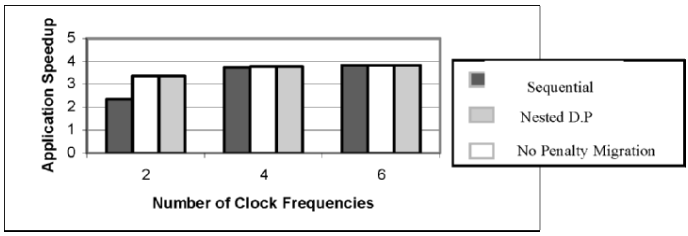


Figure 2. Results of the heuristics on a commercial quality video decoder. Compared to a single-frequency, single-coupling implementation of the accelerators, the heuristics improve the execution time by almost 4x.

To further test our heuristics, we applied our heuristics to several synthetic examples, which included a wide range of accelerators. Each example supported a large range of computation cycles, memory accesses, and clock frequencies. Figure 3 highlights results of comparing the *No Penalty Migration* and *Nested Dynamic Programming* heuristics to an implementation that did not consider coupling or multiple clock frequencies. Figure 3(a) shows the benefit of just including one additional clock frequency, and thus introducing the ability to tightly or loosely couple each accelerator. With only two clock frequencies, Figure 3(a) shows the heuristics are able to achieve on average 4x speedup. Note that in every case the *Nested Dynamic Programming* heuristic finds the best partitioning of the accelerators. The *Nested Dynamic Programming* heuristic also took the longest to complete, finishing many seconds later in the larger examples. The *No Penalty Migration* heuristic yielded an average 15% improvement in application running time over the straightforward sequential approach. The *Nested Dynamic Programming* heuristic gained an additional 15% improvement over *No Penalty Migration*. This was because both the sequential search and *No Penalty Migration* partitioned several accelerators to the tightly coupled set without knowledge of the fact that there were only

two clock frequencies available. The *Nested Dynamic Programming* heuristic was able to test all combinations of accelerators in the tightly coupled set, and therefore was able to find a superior solution.

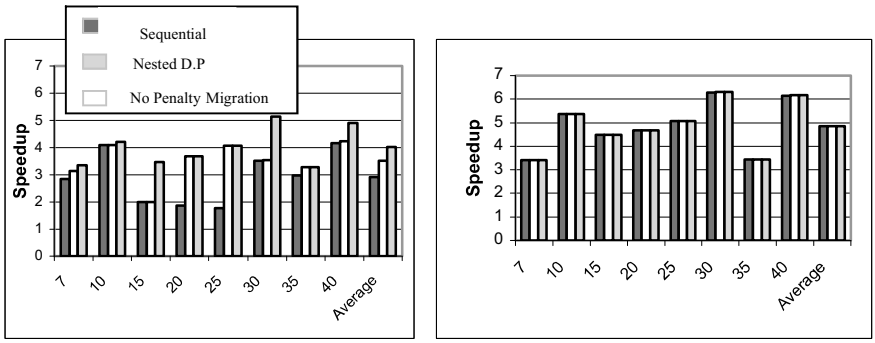


Figure 3. Application speedups for synthetic examples with varying numbers of accelerators: (a) two clock frequencies, (b) eight clock frequencies. Substantial speedup is achieved for increasing numbers of clock frequencies compared to single-frequency, single-coupling implementations.

However, as the number of clock frequencies increased, the heuristics achieved nearly the same speedups. The reason is because as the number of available clocks increase, it is more likely that the initial partitioning of the tightly coupled set is correct, meaning only minor gains could be made over a straightforward sequential search. On average across 2 to 6 clock frequencies, *No Penalty Migration* yielded a 5% improvement over a sequential search, while *Nested Dynamic Programming* provided a 10% improvement. Comparing Figures 3(a) and 3(b), one sees that additional available clock frequencies does improve speedups over single-coupled single-frequency partitions, from an average of 4x in (a) to nearly 5x in (b), with one example achieving almost 6.5x performance improvement.

For all the examples, our heuristics ran in seconds, compared to an exhaustive search, which did not complete in any reasonable amount of time when the number of accelerators exceeded fifteen. The *No Penalty Migration* heuristic completed its search consistently faster than *Nested Dynamic Programming* heuristic while also finding a better solution for platforms with only a few available clock frequencies. However, the *Nested Dynamic Programming* heuristic might be much easier to implement in a framework where coupling and clock assignments have already been implemented.

5. CONCLUSION

We showed that the consideration of both coupling and multiple clock frequencies can lead to substantial speedup over an application implementation that does not consider either. We also showed that the integration of both coupling and multiple clock frequencies can lead to application speedups of over 5x compared to a single-coupling single-frequency implementation. We developed two new heuristics that integrated coupling and clock frequency assignment, running in just seconds.

ACKNOWLEDGEMENTS

This work was supported by grants from the National Science Foundation (CNS-0614957) and the Semiconductor Research Corporation (2005-HJ-1331), and by donations from Xilinx, Inc. Freescale Semiconductor supplied the commercial H.264 decoder

REFERENCES

1. Gupta, R. and G. De Micheli. Hardware-Software Cosynthesis For Digital Systems. IEEE Design and Test of Computers. Pages 29-41, September 1993
2. Henkel, J. A low power hardware/software partitioning approach for core-based embedded systems. In Proceedings of the 36th ACM/IEEE Design Automation Conference, 122–127, 1999.
3. Corp. 2005. FPSLIC (AVR with FPGA), http://www.atmel.com/products/FPSLIC/.
4. Virtex II and IV. Xilinx Corp., http://www.xilinx.com
5. Hu, J., Y. Shin, N. Dhanwada, and R. Marculescu. Architecting Voltage Islands in Core-Based System-on-a-Chip Designs. Int. Symp. on Low Power Electronics and Design (ISLPED), 2004, pp. 180-185.
6. Sirowy, S., Y. Wu, S. Lonardi, and F.Vahid. Two-Level Microprocessor-Accelerator Partitioning. Design and Test Europe(DATE) 2007.
7. Sirowy, S., Y. Wu, S. Lonardi, and F. Vahid. Clock-Frequency Assignment for Multiple Clock Domain Systems-on-a-Chip. Design and Test in Europe(DATE). 2007.
8. Lengauer, T. 1990. Combinatorial Algorithms for Integrated Circuit Layout. John Wiley & Sons, Inc., New York, NY.
9. Stitt, G., F. Vahid, G. McGregor, B. Einloth Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS), Sep. 2005

EMBEDDED VERTEX SHADER IN FPGA

Lars Middendorf, Felix Mühlbauer[1], Georg Umlauf[2], Christophe Bobda[1]

[1]*Self-Organizing Embedded Systems Group, Department of Computer Science*
University of Kaiserslautern

[2]*Geometric Algorithms Group, Department of Computer Science*
University of Kaiserslautern

lmid@gmx.de, (muehlbauer,umlauf,bobda)@informatik.uni-kl.de

Abstract: Real-time 3D visualization of objects or information becomes increasingly important in everyday life e.g. in cellular phones or mobile systems. Care should be taken in the design and implementation of 3D rendering in such embedded devices like handhelds devices in order to meet the performance requirement, while maintaining power consumption low. In this work, the design and implementation of a vertex shader on a reconfigurable hardware is presented. The main focus is placed on the efficient hardware/software partitioning of the vertex shader computation, in order to maximize the performance while maintaining a high flexibility. The resulting solution must be compatible to existing vertex shaders in oder to allow the large amount of existing program to be easylly ported to our platform. A prototype consting of a PowerPC, peripherals and some custom hardware modules is realized a on an FPGA-board. The implementation of a point rendering shows considerable speed up compared to a pure software solution.

Keywords: 3D graphics, vertex shader, Direct3D, embedded systems, FPGA

1. INTRODUCTION

Rendering of three-dimensional objects in real-time requires much arithmetic performance. This is a problem for embedded systems that are running at low clock speed and often lacks dedicated hardware processing modules like a floating point unit (FPU). In desktop computers, the expensive arithmetic computations related to the rendering of 3D objects are done by specialized stream processing hardware in video cards. Those cards are programmable using small programs called *shaders*. The execution of shaders is the main difference to the CPU. A new instance of the program is invoked for every primitive, vertex or pixel. There are three slightly different types of shaders

Middendorf, L., Mühlbauer, F., Umlauf, G., Bobda, C., 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 155–164.

for these elements. Each instance can be executed independently of the others because there is no communication possible between instances of the same type. This is advantageous when designing the hardware, because it allows the execution of an arbitrary number of instances in parallel, in order to gain the maximum computation speed. Also, pipeline technique can be used to allow some threads to feed parameters to other threads waiting for them in the pipeline. As a result the available hardware can be used more efficiently.

We developed a hardware accelerator for executing vertex shaders that is in particular useful for embedded systems, because it uses very few hardware resources, in this case FPGA slices. It is a kind of coprocessor that is directly connected to the CPU by a fast bus. The main program running on the CPU loads the shader code and all inputs into this coprocessor. While the coprocessor is running the shader, the main program accomplishes further computations in parallel until the results can be read back.

It is important to minimize the resource usage of the hardware, because the number of available slices in a FGPA is very limited. The clock speed is also very low and we have to maximize the utilization of sub-components in all cycles. The scheduling of the threads is therefore pre-calculated and stored as part of the shader code. Hence, the control-logic consists only of the program counter and a few multiplexers, that route the data flow, thus enough space is left on the device to implement floating point calculations. The multiplexer configuration is stored in a table with one row for each cycle. In this work a shader converter that generates the control table from a Direct3D9[4] vertex shader was developed. The shader converter performs the scheduling of all operation on the generated hardware unit, analyzes and optimizes the data flow and maps the calculations to operations of our ALU. Currently we can execute four threads on the ALU in parallel. This allows a speed-up of factor four compare to the software implementation of the shader.

The rest of the work is organized as follows: Section 2 provides the basics of vertex shader while section 3 introduces some work related to custom implementation of vertex shaders. Section 4 explain our implementation. A naive co-design approach is first explained, followed by a more efficient one. Also the design decisions for the hardware software partitioning are explained. The results obtained on a prototype implemented on a Xilinx Virtex 4 evaluation platform are given in 5. Finally section 6 concludes the work and provides some indication on the future directions.

2. VERTEX SHADER

In a rendering process, each 3D-point, also called *vertex* must traversed a set of computing stations, the *render pipeline* until the final step wher it can be drawn on the display. The stations consist of a coordinate transformation

(object → world, world → camera) stage, an illumination, a clipping, a projection, a scaling to screen resolution step, and finally the step to approximate the float values to integer values is performed.

To simplify spatial calculations in computer graphics *homogeneous coordinates* (x, y, z, w) are used. Transformations like *translations*, *rotations*, *scalings*, *shearings*, *projections*, etc. can be mapped to 4×4 matrices and can be combined to only one matrix by multiplying the corresponding matrices. Thus, a transformation of a vertex by a certain list of transformations can be realized by one matrix-vector-multiplication.

For illumination calculations the dot product (scalar product) is very important, because the light intensity depends on the angle between surface normals and light sources. Normals can be transformed similar to vertices which is advantageous when filling the surface normals together with the vertices of the scence into the render pipeline to speed up processing.

In conclusion, each stage of the render pipeline executes mainly matrix and vector operations using all values which are involved like coordinates, surface normales, surface attributes, lightning parameters, etc.

There are several vertex shader versions for different hardware. We focus on implementing a subset of the smallest version 1.1 [4]. All versions use a RISC instruction set. Each instruction can read from up to three registers and write to one result register. Almost every register is 128 bits wide and stores four 32 bit floating point numbers. Hence most of the commands operate on vectors with four components. The individual components of a vector can be reordered and duplicated while reading from a register and there is a write mask for every component of the result register. This improves flexibility and allows optimizing calculations. It is for example possible to get a cross product with two instructions. Because our hardware is scalar-based, all write and swizzle-masks are free and should be used to improve performance. The shader converter analyzes the data flow for each individual component and if a result is not used, the calculation is removed on a per-component basis.

There are global and local registers. Each instance of the shader has its own set of local registers consisting of temporary and output registers. It is not allowed to write global registers which makes parallelizing possible, because there is no synchronization required and no operation in one thread depends on results calculated in another thread.

Vertex Shader 1.1 does not support jumps or subroutines. A detailed description of the instruction set can be found in the DirectX SDK [4].

3. RELATED WORK

Lots of work has been done already in the domain of accelerating graphics applications utilizing FPGAs in general. Some of them are listed in [8, 6]. Often

ten a combination of a desktop computer and a FPGA builds the computing unit. The need of 3D graphics visualization in embedded systems is still growing with the increasing spreading of mobile multimedia systems in everyday life like cellular phones and PDAs. Even the MPEG H.264 standard which is the video coding for next-generation multimedia involves rendering of 2D and 3D deformable mesh geometry [5].

The still continuing miniaturization has led to highly integrated chips and finally to so-called SoCs (system on chip). Here all components and peripherals are placed on a single chip like processors, hardware accelerators, bus and peripheral controllers and allow a PLB (printed circuit board) independent redesign or update of applications which is an important advantage.

Sohn et al. introduced a multimedia co-processor for mobile applications using an ARM-10 processor and fixed-point arithmetic [1]. The company Bitboys developed an vector graphics processor targeting for high-end multimedia cellular phones which is available as IP core for SoCs integration and can process SVG and OpenVG object data [7].

We are particular interested in a system in which custom hardware can cohabit with software. Also, the system should provide enough flexibility to ease the redesign and also allow a run-time adaptation, while maintaining the performance high and the power consumption low. The next sections explain our solution to this problem.

4. IMPLEMENTATION

Our target platform in this project was a Xilinx Virtex 4 evaluation board featuring a Virtex4-FX12 FPGA. This FPGA contains an embedded PowerPC 405 processor, on-chip memory (BlockRAM) and miscellaneous DSP functions. We use the external DDR-RAM as video frame buffer to store 3D object data. A simple system on chip with DDR-RAM controller, VGA out module and system bus needs already half of the available slices of the FPGA. Because, floating point hardware modules are expensive, we tried to avoid or reuse them as much as possible. Thus, an efficient design considering speed and chip area has to be found.

Basic Design

In a first design a field of 32 registers combined with an adder and a multiplier unit and an instruction memory was drawn up. This *co-processor* is directly connected to the main processor via the FCM bus, which allows to extend the native PowerPC instruction set with custom instructions that are executed by a user-defined configurable hardware accelerator.

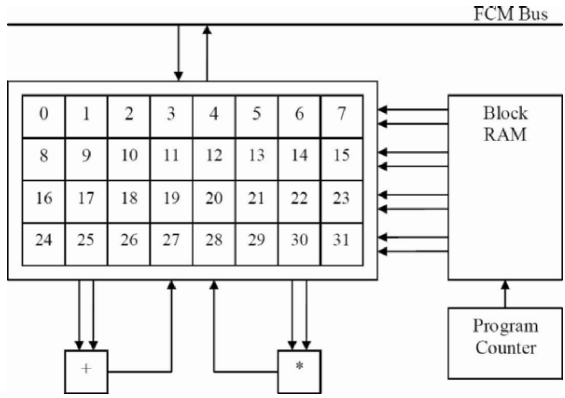


Figure 1. First design: Field of registers

The data words read from the BlockRAM (see Figure 1) specify which registers supply the input values for the arithmetic units and to which register each result should be written back.

The implementation of this design is very straight forward and also expandable for further operations like division or square root. So the two(or more) operations are executed simultaneously. Unfortunately, the design needed huge multiplexers and address decoders leading to very high resources consumption. The complete chip area was filled by this first version of the design.

Final Design

In order to improve the first design, the idea was to exchange the expensive registers, previously realize using the on-chip available logic (LUTs) to on chip memory, namely dual ported BlockRAM. These can hold up to 512 values each (compared to 32 for the register field) but only two read respectively write accesses are possible simultaneously. Because the dot product needs 8 input values and since only one value can be provided by a BlockRAM, we duplicated data to 8 BlockRAMs in order to be able to read eight values simultaneously. In order to keep the consistency in all 8 BlockRAMs all write request are dispatched to all 8 BlockRAMs (Figure 2). In the following this BlockRAM unit is called *register array*. This new design consumes very few slices and also provides much space for provisional results. Compared to registers a memory read access takes one clock cycle and could cause additional delays in the computation. However, due to the saving of slices, an efficient design of the ALU will compensate the lost in the BlockRAM usage (Figure 3).

We next explain the components of the final design (FCM Controller and ALU) in more detail.

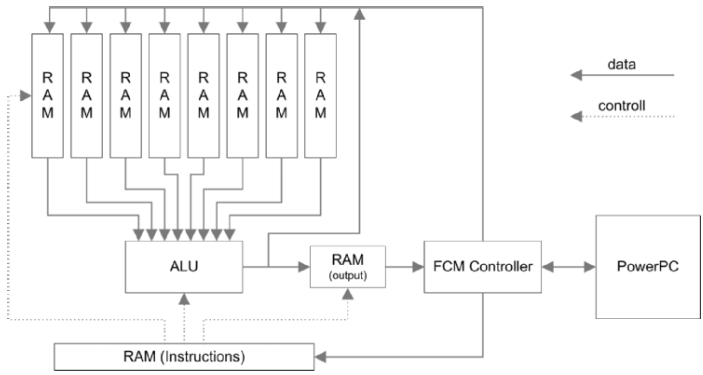


Figure 2. Final design

FCM Controller. The control module implements the interface to the FCM bus. The CPU is able to write to the register array, which can hold up to 128 4D vectors, and to the instruction memory with a maximum of 512 opcodes. The FCM controller is also able to read back the results from the output memory. Because the FCM instructions to handle double words provide only 5 address bits an additional 2 bit register is used to access all 512 possible memory locations.

The final result of the shader is stored in a special output RAM that is written by the ALU and read by the CPU and the FCM control module. The output RAM is addressed independently from the register array which allows copying the shader results to an arbitrary position. The additional RAM also saves one multiplexer that, otherwise, would be needed at the address lines of the register array to switch between the ALU and the control module.

ALU. The ALU has eight floating-point input variables, one input port that is used to select the equation, and one output port which can be used to get the result as shown in Figure 3.

In every cycle there are nine different outputs available. One of them is selected by the output multiplexer and controlled by the current instruction code. Some results are intermediate result of longer calculations and have therefore a shorter latency. Table 1 lists the instruction set of the ALU. The arithmetic units use pipelining to save hardware resources and cause delays which are also shown in the table. When generating instructions for the ALU this behaviour has to be taken into account. Still, the ALU can accept one set of values per clock cycle.

Every input variable can be pre-multiplied by -1 before it is read into the ALU. This is implemented as an exclusive-or between the sign bit of the floating-point number and the corresponding instruction bit. Because of de-

lays the `slt` instruction that is used for comparisons, minimum, maximum and absolute value, the parameters e, f, g, h must be provided twice. The `rsq` command returns a rough approximation for the inverse of square root which is much more likely to be used as the square root itself, e.g. for normalization of vectors. Usually for a more precise result one step of the Newton iteration (formula: $x_{n+1} = \frac{1}{2}x_n(3 - x_0{x_n}^2)$) is sufficient [3].

Table 1. ALU commands

command	*result*	*delay*	*notes*
dot4	$a \cdot b + c \cdot d + e \cdot f + g \cdot h$	19	4D dot product
dot2	$a \cdot b + c \cdot d$	14	2D dot product
mult4	$a \cdot b \cdot c \cdot d$	18	multiplication
mult2	$a \cdot b$	9	multiplication
div	a/b	27	division
rsq	0x5F3759DF $-(a \gg 1)$	2	start value for newton iteration of $\frac{1}{\sqrt{a}}$
slt	if $(a \cdot b + c \cdot d < 0)$ then $(e \cdot f)$ else $(g \cdot h)$	14	input values are needed after 5 clock ticks again
int2float	$float(a)$	6	converts integer to float
float2int	$int(a)$	6	converts float to integer

Instruction Format. All instructions have a fixed length of 128 Bit, because of the eight input registers. The whole instruction can be divided into four words with the layout shown in Table 2. The input values are read from memory at position `src*` and inverted according to `inv*`. The ALU result is stored at index `dst`, if `we` (write enable) is set and in the extra output RAM if `oe` (output enable) is set. To avoid an extra function de-multiplexer for each ALU command a selection bit was arranged (see remaining entries in Table 2).

Table 2. Instruction format

Word	*0:8*	*9:17*	*18:26*	*27*	*28*	*29*	*30*	*31*
cmd0	src0	src1	dst	we	inv0	inv1	inv2	inv3
cmd1	src2	src3	out	oe	inv4	inv5	inv6	inv7
cmd2	src4	src5	-	div	rsq	slt	mult2	dot2
cmd3	src6	src7	-	f2i	i2f	mult4	dot4	-

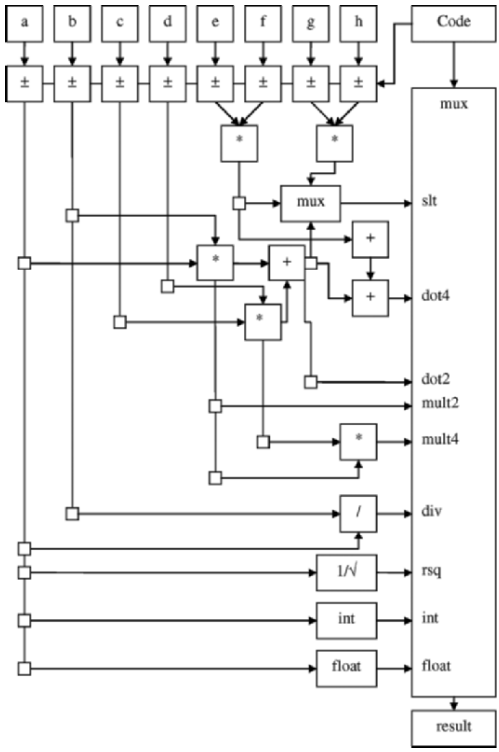


Figure 3. ALU

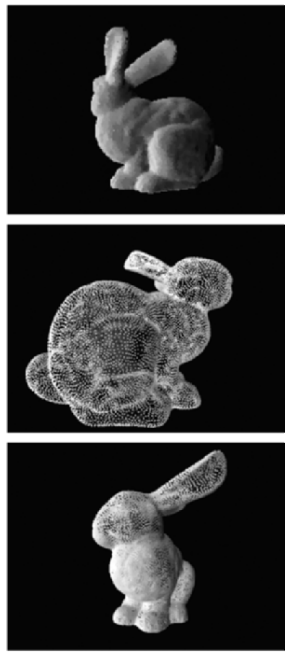


Figure 4. This bunny consist of approximately 20.000 vertices [2]

Vertex Shader Converter

To generate the ALU opcodes a given vertex shader program is compiled with DirectX SDK[4]. Using the syntax analysis for the resulting code a data flow graph is build up, which points out the dependencies between input, provisional and output values. Now, vector operations are mapped to scalar (ALU) operations and long processing chains are move to the program start, while considering the delays caused by the arithmetic sub-units. Multiplications by -1 can be handled directly by the ALU input stage. Diversions which are not available in *Vertex Shader 1.1* and therefore are realized by multiplication with the inverse, can be processed directly by the ALU. Sometimes algebraic conversions can help to map calculations to the optimized dot product (e.g. $(a + b)c \rightarrow ac + bc$). Also the usage of the `slt` command is more practical.

5. RESULTS

The most important disadvantage of this implementation is the limitation to one result per cycle. This means that a matrix-vector multiplication takes at least four cycles. The high latency of certain operations is not really a problem, but it is different for almost every instruction, so that it can be difficult or impossible to fully load the ALU. Because of the strict requirements, not all commands could be directly implemented in hardware. For example a single `mul` instruction that multiplies two vectors component wise can take up to four cycles. But usually the instruction is part of a more complex calculation and the shader converter can merge the previous and following calculations so that the whole block may be mapped to four larger instructions that also take four cycles.

On the other hand the ALU can calculate a 4D dot product every cycle. It has been chosen to be specially optimized, because it is a very important and often used operation. Even the most simple but useful shader does a vector-matrix multiplication that can be calculated using four 4D dot products. A large number of other instructions using only multiplications and additions can be reordered and mapped to dot products. But the most important reason for the dot product is the fact that it has only one scalar result and fits perfectly to the limited register array that can only write one result value. It is also slightly cheaper than a parallel componentwise multiplication and addition because it only needs three addition modules.

There is the possibility to output directly the intermediate 2D dot product and to skip the last addition for a lower latency. This can be useful when interpolating between two vectors. The additional multiplier outside of the dot product gives the ability to multiply four floating-point numbers in one cycle. This is important for calculating multi-linear functions that could otherwise only be achieved by a large number of cumbersome repeated high latency dot products.

This design cannot be enhanced any further. Adding another instruction type extends the multiplexer at the output of the ALU and leads to increased complexity. The timing constraints will not be met and the required clock speed of 100MHz cannot be achieved.

The new hardware component has been tested with a mesh viewer. The viewer is running on the PowerPC CPU, but the vertex shader can be calculated either in software or hardware to compare the performance. The triangles are not filled and the mesh is rendered as a point model (bunny model from [2]). We want to measure the speed of the vertex calculations and in a real application the expensive triangle filling would also be done in hardware. For each configuration 100 frames have been rendered several times with different point counts. The time spans are very precisely measured directly on the board

with a special 64 bit register that counts the CPU cycles. The vertex shader consists of six instructions that calculate the coordinates and the lighting from a directional light source.

Comparing the results both for software and hardware it is obvious that the hardware accelerated version is much faster, see Table 3. The last column of Table 3 contains the ratio between software and hardware performance. The ratio is higher when rendering more vertices, because there is a fixed overhead per frame for clearing the color and depth buffers.

Table 3. Performance results for 100 frames [sec].

Vertex Count	*Hardware*	*Software*	*Software/Hardware*
5000	3.607s	13.32s	3.693
10000	4.832s	24.25s	5.019
15000	6.104s	35.22s	5.770
20000	7.409s	46.26s	6.244

6. CONCLUSION

We have introduced a hardware accelerator for a vertex shader. Our design consumes few resources (slices) on FPGA, while supporting almost all functions of the common language for such data processing *Vertex Shader 1.1*. Compared to a software only version a significant speed advantage could be achieved. This application is suitable for the domain of embedded systems.

REFERENCES

[1] Jerald Yoo Ju-Ho Sohn, Jeong-Ho Woo and Hoi-Jun Yoo. Design and test of fixed-point multimedia co-processor for mobile applications. In *DATE 2006*, 2006.

[2] Leif Kobbelt. Special effects SS05, 2005. `http://www-i8.informatik.rwth-aachen.de/old-site/teaching/ss05/praktikum_sfx/` [date: 09/12/2006].

[3] Chris Lomont. Fast inverse square root, 2003. `http://www.math.purdue.edu/~clomont/Math/Papers/2003/InvSqrt.pdf` [date: 2003].

[4] Microsoft Corporation. Directx sdk, 2006. `http://www.microsoft.com/directx` [date: 09/12/2006].

[5] Iain Richardson. *H.264 and MPEG-4 - video compression*. Wiley, 2003.

[6] Henry Styles and Wayne Luk. Customising graphics applications: Techniques and programming interface. In *IEEE Symposium on Field-Programmable Custom Computing Machines 2000*, 2000.

[7] symbian.com. Bitboys introduces vector graphics processor for mobile devices at game developers conference. *www.symbian.com*, 2005.

[8] David Thomas and Wayne Luk. *Implementing Graphics Shaders Using FPGAs*, page 1173. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004.

A HYBRID APPROACH FOR SYSTEM-LEVEL DESIGN EVALUATION*

Alexander Viehl[1], Markus Schwarz[1], Oliver Bringmann[1]
and Wolfgang Rosenstiel[1,2]

[1]*FZI Forschungszentrum Informatik*
Haid-und-Neu-Strasse 10-14
76131 Karlsruhe
[viehl,schwarz,bringman]@fzi.de

[2]*Universität Tübingen*
Sand 1
72076 Tübingen
rosenstiel@informatik.uni-tuebingen.de

Abstract: We present a novel approach for performance and power analysis of a system design. For that purpose, we combine coarse grain profiling and established analysis models in a hybrid approach for an early qualitative determination of system properties. Our approach considers the control-flow of communicating processes and the impact of blocking communication on the temporal behavior of the entire system. This can be used for the determination of the system performance and the consideration of the performance impact of dynamic power management. Based on this, the energy consumption of a system can be estimated. The application of our new methodology leads to an inclusion of probabilities concerning system properties and allows an early risk analysis of a design relating to predefined system requirements and constraints.

Keywords: Performance Analysis, Power Estimation, Risk Evaluation

1. INTRODUCTION

Early available design characteristics are essential for designers to validate a design with respect to requirements. Whereas different approaches exist to incorporate worst-case and best-case properties of the system, the resulting intervals (for e.g. performance [12, 5]) are often too pessimistic and distant from

*This work was partially supported by the BMBF project URANOS under grant 01M3075D and by the DFG project "Communication Analysis for Network-on-Chip" under grant BR 2321/1-1

Viehl, A., Schwarz, M., Bringmann, O., Rosenstiel, W, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 165–178.

the real behavior of the system. Moreover, rare cases that appear only once like initial cache misses can expand the intervals of quantitative analysis too much. A resulting architecture of a system concerning these worst-case analysis is quite often overdimensioned. Although this might be acceptable for real-time critical systems, the designer does not know about the typical qualitative and quantitative behavior at an early design stage. Especially in the domains of multimedia and telecommunication as well as real-time embedded systems, there is a demand for early statements about the typical behavior of the system relating to requirements from the specification. In these areas, estimations are sufficient for an early evaluation of the behavior of a design consisting partitially of qualified IP, in which design parameters are configuration specific – for instance mapping, clock frequencies, instruction sets and caches. Besides performance characteristics, mobile applications demand early power estimations of an application platform processing a specific input stream. Without any qualitative characterization of the system behavior, the designer does not know quantitatively how many cases are critical and not met. We present a hybrid approach for qualitative system-level design evaluation in this article. Our novel methodology tries to combine the advantages of both worlds: the derivation of qualitative expressions from profiling and the analytic incorporation without simulating the entire system. In the case of a modification of the system towards exploration, our methodology requires only the determination of the run-times of modified parts.

On one hand, we are using coarse grain profiling for determining the run-time of communication-free parts of a system model. This architectural mapping process is briefly described in Section 3. The resulting quantities are incorporated in an analytic system model, namely the communication dependency graph (CDG) presented in Section 2. Using this graph, we are performing system-level analysis to determine qualitative and quantitative statements about the system behavior. Using the run-times in the activity model we calculate an idle time distribution, which describes how long the control-flow of the processes has to wait in blocking communication nodes. If the qualitative and quantitative behavior of the system relating execution and idle times is known, this information can be used for estimating the energy consumption. For that purpose, we use an established activity driven power model that is briefly presented in Section 2. Furthermore, the drawback of wake-up times from different power states can be incorporated in performance analysis. The derived qualitative information can be further used for an exploration of the underlying system architecture. Our evaluation methodology is presented in Section 4. We applied our methodology for qualitative design evaluation on a real design of a Viterbi decoder. The results are shown in Section 5.

2. RELATED WORK

The internal system behavior and probabilities concerning timing are considered by different analysis models (e.g. generalized stochastic Petri nets (GSPN) [9], stochastic automata networks (SAN) [8] and stochastic automata [4]). These models are used to explicitly model and analyze applications and systems including the internal behavior of a system. Timed Petri Nets [16] are able to represent the internal behavior of a system. Although there exist stochastic extensions by generalized stochastic Petri nets (GSPN) [9, 11] for performance and power estimation, the model does not represent execution times of real system components. Furthermore, synchronization by communication and the specification of communication types have to be modeled explicitly and can not be extracted from executable functional specifications like a SystemC model of a design. System-level performance and power analysis based on Stochastic Automata Networks (SAN) are introduced in [8]. The system including probabilities of execution times is modeled explicitly in SAN. The real execution behavior of the components related to timing and control-flow of a functional implementation is not considered. On the other hand, profiling is used for the determination of characteristics like execution time, resource utilization or memory requirements. This approach is very fine grain and slow. Transaction Level Modeling (TLM) [6] uses models with components on different levels of abstraction for speeding up simulation with a potential loss of accuracy. Profiling or simulating a complete system of parallel processes consumes a lot of computational resources. Moreover, synchronization overhead for coupling multiple simulators is an issue.

Activity Model

To represent the temporal behavior of a system, a model called communication dependency graph (CDG) is used. It was originally developed for formal communication analysis by determining communication instances that synchronize the control-flow of communication partners [13]. We are using this model for the system-level representation of a design. The model considers temporal best-case and worst-case properties of communication and computation. We are extending the model in Section 3 to express qualitative timing properties.

A *communication dependency graph* (CDG) denotes a consolidated representation of a system consisting of communicating processes. Only the temporal and causal behavior between communication endpoints is considered in each communication process. The control-flow is represented by edges connecting communication endpoints. An edge exists if a path in the control-flow graph connects the communication endpoints without passing any other communication endpoint. The communication endpoints are characterized relating

their synchronization behavior, and whether they represent sending or receiving events.

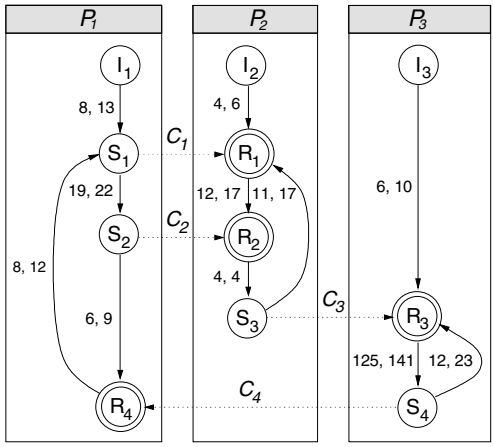


Fig. 1: Communication Dependency Graph

A CDG example consisting of three processes P_1, P_2, and P_3 is depicted in Figure 1. The processes communicate using asynchronous communication instances C_x with non-blocking sender and blocking receiver nodes.

Power Management Model

We use Power State Machines (PSM) [3, 2], to characterize the behavior of dynamic power management of the platform components. The model is flexible enough for describing arbitrary power management schemes. Nodes of

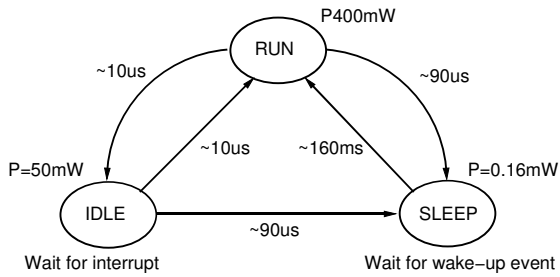


Fig. 2: PSM of SA1100

the PSM characterize different states like `IDLE` or `RUNNING`. Edges between nodes characterize state transitions relating to events. Both nodes and edges can be annotated with expressions of timing and power consumption. This

allows the expression of different properties of dynamic power management (e.g. wake-up penalties). The PSM of a StrongARM 1100 is depicted in Figure 2. It consists of three different power states. State transitions are triggered by inactivity or activation of resources due to external events. We can determine the energy consumption of platform components if a characterization of the temporal behavior of the related component is given.

3. ARCHITECTURAL MAPPING

The starting point of our mapping and exploration flow depicted in Figure 3 is a functional SystemC [10] model of a design, a platform description, mapping information, and the environment of the system. The SystemC model is

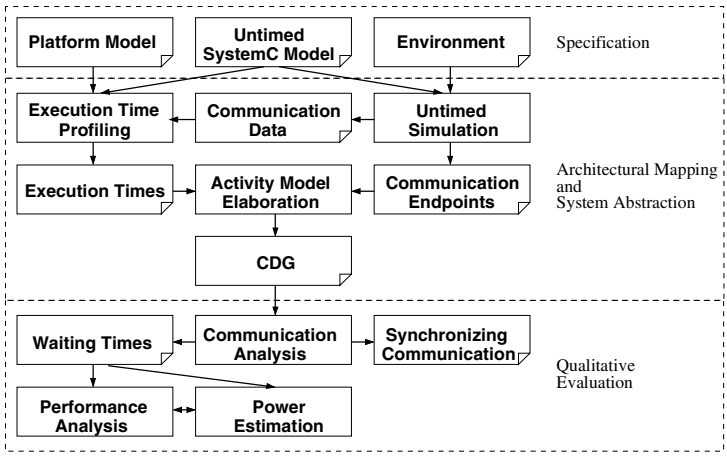


Fig. 3: Mapping and Analysis Flow

simulated together with a specified environment that defines the interaction of the world with the model (e.g. sensor packets for an automotive control system and the temporal interval between the packets). Introspection is applied for gaining access to the communication data sent between SystemC processes. The simulation determines the following two results: temporally ordered communication calls of each process and the data sent through each communication channel.

The ordered communication calls are used to elaborate a consolidated control-flow of each CDG process. The communication data are used as input for determining the run-time of code fragments on the target platform concerning the data computed by the design originating from the environment. For performing execution time profiling, each process from the original SystemC model has to be mapped to the target platform. Therefore, we are using the

tool PORTOS [7] to generate a set of C/C++ files, each including the functionality of one SystemC process. The access to SystemC ports is mapped to special `read(...)` and `write(...)` calls. We are instrumenting the code by linking these calls against communication stubs containing special system calls and assembler routines.

After compilation, each single process is executed on an Instruction Set Simulator (ISS). Therefore, the SimpleScalar [1] ISS was extended to handle the assembler routines that were added to the communication stubs. When a communication function is called, the special syscall is recognized by the extension to the ISS. In the case of write access, two timestamps are taken: one at the beginning of the routine and one at the end. Both timestamps encapsule the computation time of the communication stub. Using the first timestamp with the last timestamp of the previous communication access, the computation time between two subsequent communication of one process is calculated. It is not necessary to store the communication data because these were already collected during untimed SystemC simulation of the entire system. The stubs are implemented to have minimal impact on cache data. Hence, typical communication drivers have an impact on cache data and on execution time as well. These effects can be abstracted and included in the activity model by appending duration times to communication.

In the case of a read access, the syscall is additionally handled by inserting the data of the communication in the buffer address space of the ISS. These data were previously collected by running an untimed SystemC simulation. As a result, the same input data are used by the single processes running on the ISS in comparison to the parallel execution of the whole SystemC model.

In the next step, the collected execution times are combined with the elaborated control-flow to the activity model (i.e. CDG). The same methodology can be used to incorporate synthesized hardware blocks or IP by using HDL simulators instead of an ISS.

Based on the CDG, communication analysis is performed as described in Section 4 for determining the impact of blocking communication on the runtime of the entire system. Further results are the detection of non-synchronizing communication that indicate need for buffer insertion or structural bottlenecks in the system design.

In Section 4 we present the analytic incorporation of this information for qualitative system evaluation relating power and performance.

Execution Time Models

We use two different models of computation for qualitative design evaluation. For the calculation of the behavior concerning the real order of execution times, we extend it with an ordered vector of execution times. This means

that each particular execution time of each edge is stored. This enables analysis of the system performance without any need for determining execution times again. We assume a deterministic temporal behavior of the components relating to input data.

For our estimation approach, we assume that an edge can have n different execution times $lp_1, ..., lp_n$. These execution times appear with probabilities of $P(lp_1), ..., P(lp_n)$. The execution times and probabilities can be simply calculated from profiled execution times. Furthermore, the execution time distribution can be taken from a specification or assumed by the designer.

Activity Model Elaboration

The ordered vector of communication endpoints CE derived from untimed simulation is used as input for an elaboration algorithm. The $ce \in CE$ are characterized by unique IDs. The objective is the elaboration of a consolidated control-flow of each process. It assumes that a deterministic control-flow according to the order of communication endpoints exists. Nevertheless, the algorithm is able to incorporate branches that end at different communication endpoints. If the selection of branches are changing indeterministically, the control-flow is unrolled. If the selection of branches is deterministic, the algorithm detects a recurring state and inserts a backward edge in the elaborated process. Each communication endpoint is characterized by its blocking property, and whether it is $send$ or $receive$. It returns a set of nodes and a set of edges as a representation of one CDG process. At the beginning, new nodes are inserted as long as no other node with the same ID exists. When a node with an already existing ID is inserted, the assumption is that a backward edge to the existing node is inserted. The current state is stored. Then the next nodes from CE are compared with the existing successors of the already existing node in the CDG. If there is a deviation between the order of the nodes in CE and the already inserted nodes, the algorithm uses backtracking and restores the state that was saved before inserting the backward edge. After that, a new node and an edge connecting the node with its successor are inserted. Otherwise, if the same node from which the backward edge was created is reached, the insertion of a backward edge was correct and the state of the system after inserting the nodes from CE is stored. Later, execution times are appended to an edge of the elaborated graph using the ID of enclosing communication endpoints and the order of the communication.

4. QUALITATIVE EVALUATION

Communication Analysis

The goal of communication analysis is the determination of waiting times in blocking communication endnodes. The time the process has to wait in a blocking end node of communication instance C of the CDG is denoted by the slack variable $x(C)$. This time is calculated from the last communication instance C_{pre} that potentially synchronizes the two processes. This means that the control-flow has to wait, if it arrives at the blocking communication endpoint of a communication instance before the control-flow arrives at the non-blocking endpoint. With an iterative algorithm starting at the `init` nodes of each process, we are using already determined slack values $x(C_i)$ for the calculation of subsequent ones. The algorithm terminates, if the vector of system wide slack values recurs. The analysis has to be performed using both models of computation:

I. The determination of system characteristics by incorporating the ordered vector of particular execution times can be used for determining the same performance characteristics as determined by a simulation of the system of communicating processes. A modification of one component only demands for determining the execution time characteristics of this component if the functionality changes. In comparison to our estimation approach (II.), we need to keep the ordered vector of execution times $ls(e)$ instead of summing up the occurrence of equal execution times. Depending on the environment of each component, these data can grow up to some gigabytes for each configuration of a component. For the i-th usage of an edge in system analysis, the i-th value of the execution time vector is used.

II. The incorporation of different execution times and probabilities as distribution function for system property estimation assumes that each execution time of an edge can be followed by each execution time of a subsequent edge. Although the needed operations for communication analysis are based on basic probability theory, issues according the incorporation of waiting times and non-synchronizing communication have to be considered. Waiting times at communication endpoints can not be simply combined with each different execution time of the path leading to this node. When waiting times at blocking communication endpoints are incorporated in path calculation it has to be ensured that they are only combined with the execution times they were calculated from. Otherwise, the analysis itself would lead to a misprediction of the system behavior. Furthermore, if communication is not or not always synchro-

nizing the control-flow of the processes due to timing issues, this impact has to be incorporated as well.

Classification of Run-Times

For reducing computational effort of the estimation approach, different run-times can be classified in groups. Although this may lead to a loss of accuracy, it provides an efficient method for deriving an initial overview about the qualitative system behavior. Arbitrary classification algorithms can be used. In this article, we divide the run-time spectrum of an edge in equidistant intervals as example. This means that all execution times in each interval are combined to one run-time and weighted depending on their probabilities. The probabilities are added. If each interval has a size of w, the maximum number of operations m for e.g. combining the latencies of subsequent paths is m/w^2. Figure 4

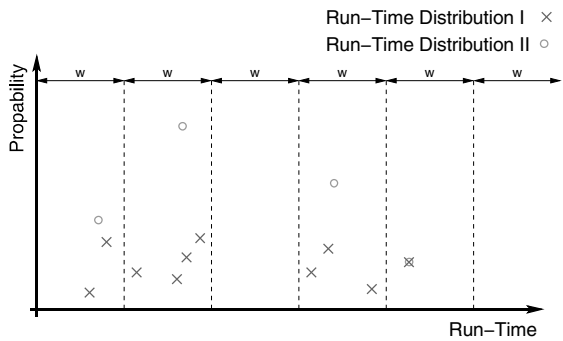


Fig. 4: Classification of Run-Time Distributions

shows a classification example. The run-time distribution II was derived from run-time distribution I using equidistant classification with intervals of size w.

Performance Analysis

Our novel approach for qualitative performance analysis is based on an approach for quantitative system analysis [13]. Performance analysis uses the information on waiting times in blocking communication endnodes that are determined by communication analysis. One direct result from communication analysis is the identification of permanently non-synchronizing communication. This leads to data loss due to high input rates. The probability of data loss can be determined using our methodology as well as the impact of buffer insertion. These information can be used for detection of bottlenecks and exploration of the underlying system architecture. We describe the qualitative evaluation of properties with regard to previously published analysis methods,

like e.g. the determination of worst-case response time (WCRT), system latency, component utilization or access conflicts on shared communication resources [14].

Power Estimation

If a dynamic power management specification is given by a PSM, the impact of wakeup-times on the performance of a system can be determined. If the PSM is not triggered by any other event, power management is activated when the system is idle. We calculated idle phases of processes waiting on a communication C as $x(C)$ by communication analysis. The potential drawback of wake-up times on the activation of processes can be expressed by $\varphi_{PSM}(x(C))$. We have to include these additional waiting times in our method for calculating subsequent slack variables. According to Section 4,

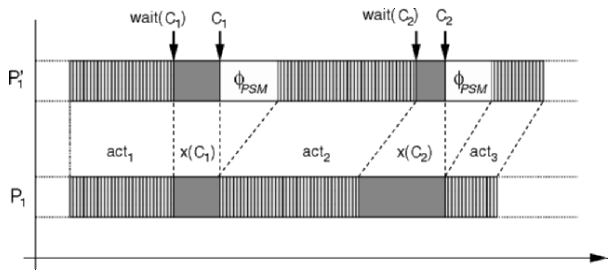


Fig. 5: Run-Time Impact of Dynamic Power Management

$|\varphi_{PSM}(x(C))|$ is added to each path that traverses a communication C or its blocking communication endpoint. For determining the energy consumption, we apply the PSM on the information about computation times stored in the CDG and on the information about $x(C)$ and $\varphi_{PSM}(x(C))$. Figure 5 depicts an example of a process $P1$ without and $P1'$ with dynamic power management. Although the activity times act_x in both examples are identical, the waiting times $x(C)$ and the impact of the PSM defined by $\varphi(x(C))$ are different. The reduced performance of this process may lead to a need for buffering, the violation of deadlines or data loss.

Architectural Exploration

Our analysis does not only allow to determine the characteristics of computation ressources. We can use the information about communication and the temporal relation between them to determine the impact of the communication infrastructure on system performance and power characteristics if typical parameters like latency, duration of communication, and a specification of

activity-based power management are given. Our approach allows an exploration towards the functionality, the environment, the mapping and the underlying platform. A modification of components or an exchange of complete subsystems can be incorporated without completely analyzing the system again. This leads to a structured reuse of qualified IP in platform based design.

5. EXPERIMENTAL RESULTS

Experimental results from the application of our methodology are presented in this section. All analysis steps were implemented and integrated into the SysXplorer [15] tool. We started with a SystemC model of a Viterbi decoder and mapped the two processes with the main functionality, `vit_fwd` and `vit_bwd` each to a PowerPC 604 with 100 MHz according to the flow presented in Section 3. The resulting elaborated CDG representing the communication structure of the communicating processes is depicted in Figure 6. The edge weights characterizing the latencies of the memories were annotated

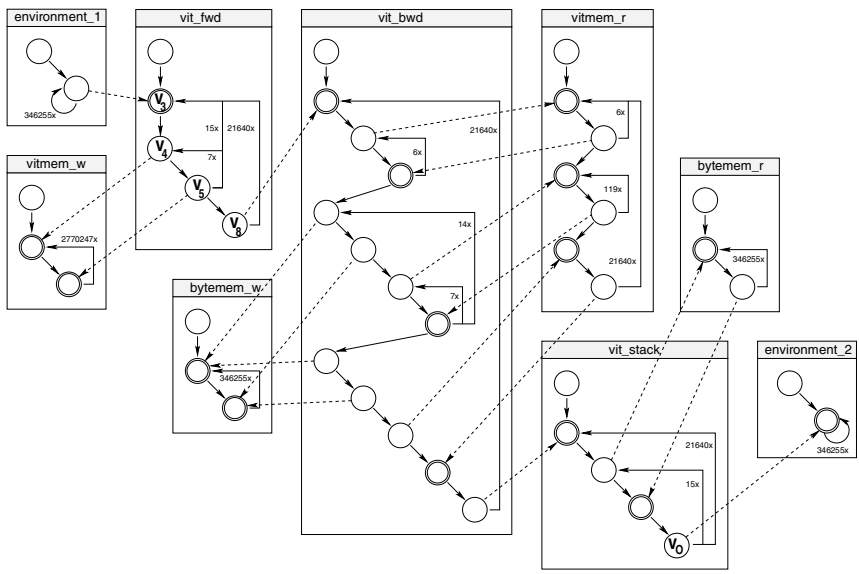
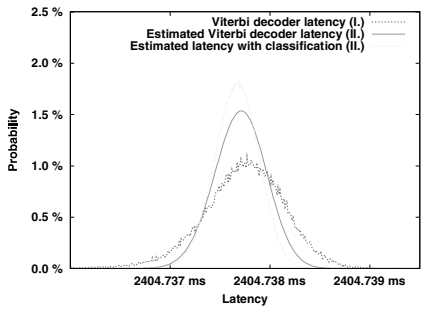


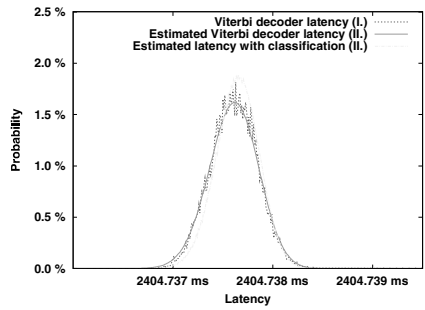
Fig. 6: Viterbi Decoder CDG

with 60 ns from the memory specification. The same latencies were used for the module `vit_stack` that has the same latency characteristics like memory. We profiled the run-times relating to two different environments the decoder processed. We used a 1 MB pgm image as one input and a 320 KB XML file as another. The data packages arrived with an inter arrival time of 160.3 ms. Of course, this arrival time model can be modified using probabilities and inter-

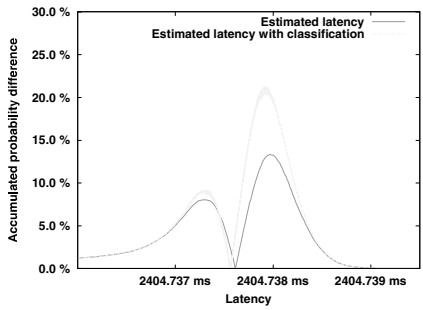
vals. Then we used the information derived from profiling and calculated the latency of the entire design between the nodes v_3 and v_O using both models of computation introduced in Section 3. A closer inspection on a comparison between particular packet latency calculation, latency estimation and latency estimation with weighted classification is depicted in Figure 7a and 7b. The classification uses equidistant intervals of 50 ns. For a comparison of the three results, we subtracted the accumulated values of both estimated curves from the particular packet curve. These results are depicted in Figure 7c and 7d. Although the estimated curve with classification does not exactly match the calculated curve using the particular execution times, it delivers a quite good approximation relating the small derivation in absolute numbers. We applied



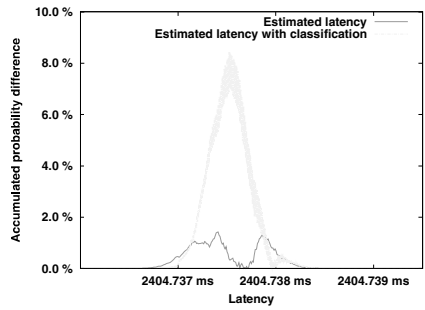
(a) 1 MB pgm (1)



(b) 320 KB XML (2)



(c) Cumulated Difference (1)



(d) Cumulated Difference (2)

Fig. 7: Comparison of Accuracy

the PSM depicted in Figure 2 to both cores. The comparison of the design latencies with and without dynamic power management is depicted in Figure 8. We used the data annotated in the PSM to estimate the energy consumption of both processor cores for computing 16 packets. The transition costs and wake-up penalties defined by the PSM reduced the impact of variable run-times on different results in energy estimation because the activation request appeared while PSM transition to `IDLE` (`vit_fwd`) and to `SLEEP` (`vit_bwd`).

		Energy Consumption (mWs)		**Latency** (ms)
		`vit_fwd`	`vit_bwd`	($v_3 \rightsquigarrow v_o$)
w/o	DPM	1025.91	1025.92	2404.73
with	DPM	1025.31	65.43	2725.13

Fig. 8: Analysis Results

Furthermore, estimation results for energy consumption are given in Figure 8. Dynamic power management of the core executing `vit_fwd` only marginally improves the energy consumption but the system latency increases due to wake-up penalties. A closer inspection on the analysis results determined that the activating input events occur after the power state of the core started transition between `IDLE` and `SLEEP`. This means that the energy intensive wake-up procedure started just after beginning the transition. Due to this, the energy consumption is still at a high state and wake-up penalties reduce the system performance. The internal control-flow of `vit_fwd` sends one resulting output packet to `vit_bwd` from 16 received input packets.

Due to this, arrival-time intervals of the communication instance between `vit_fwd` and `vit_bwd` are much longer. As consequence, the PSM of `vit_bwd` stays longer in the energy efficient state `SLEEP`. The average energy consumption of the core executing `vit_bwd` is reduced by $93.6\ \%$ in comparison to the configuration without power management. Nevertheless, PSM wake-up penalties increase the latency of the component and its performance falls off. Finally, the latency of the whole decoder is $13.3\ \%$ higher than without dynamic power management.

6. CONCLUSION

The early availability of quantified qualitative design characteristics for comparing system requirements with the design properties is valuable for designers. We developed a hybrid approach for incorporating run-time profiling and system-level analysis techniques targeting this issue. Two different models of computation were presented: One for incorporating each particular execution time from profiling and one that combines all execution times to a distribution function. Our analysis approach is based on the determination of communication that synchronize the control-flow of communicating processes. Based on waiting times in blocking communication endnodes, we calculate qualitative performance and power properties of the system. We implemented our analysis methods and presented experimental results from a real-world design, a Viterbi decoder. The comparison of both approaches shows, the hybrid estimation approach allows quite accurate requirement evaluation at system-level. Next steps will be the analytic exploration of power management policies based on

the results of communication, performance and power analysis with regard to predefined constraints.

REFERENCES

[1] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. pages 231–248, 2002.

[3] Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 173–178, New York, NY, USA, 1998. ACM Press.

[4] Jeremy Bryans, Howard Bowman, and John Derrick. Model checking stochastic automata. *ACM Trans. Comput. Logic*, 4(4):452–492, 2003.

[5] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Proceedings of DATE*, Munich, 2003.

[6] Adam Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04*, New York, NY, USA, 2004. ACM Press.

[7] Matthias Krause, Oliver Bringmann, and Wolfgang Rosenstiel. Target Software Generation: An Approach for Automatic Mapping of SystemC Specifications onto Real-Time Operating Systems. *Springer: Design Automation for Embedded Systems*, 2007.

[8] R. Marculescu and A. Nandi. Probabilistic application modeling for system-level performance analysis. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 572–579, Piscataway, NJ, USA, 2001. IEEE Press.

[9] Marco Ajmone Marsan, Gianni Conte, and Gianfranco Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, 1984.

[10] Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf, editors. *SystemC: methodologies and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[11] Qinru Qiu, Qing Wu, and Massoud Pedram. Dynamic power management of complex systems using generalized stochastic petri nets. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 352–356, New York, NY, USA, 2000. ACM Press.

[12] Simon Schliecker, Matthias Ivers, and Rolf Ernst. Integrated Analysis of Communicating Tasks in MPSoCs. In *CODES+ISSS '06*. ACM Press, 2006.

[13] Axel Siebenborn, Oliver Bringmann, and Wolfgang Rosenstiel. Worst-case performance analysis of parallel, communicating software processes. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, 2002.

[14] Axel Siebenborn, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. Control-Flow Aware Communication and Conflict Analysis of Parallel Processes. In *Proceedings of the 12th Asia and South Pacific Design Automation Conference ASP-DAC 2007, Yokohama, Japan*, 2007.

[15] SysXplorer Home. www.fzi.de/sim/sysxplorer.html.

[16] A. Yakovlev, L. Gomes, and L. Lavagno. *Hardware Design and Petri Nets*. Kluwer, 2000.

AUTOMATIC PARALLELIZATION OF SEQUENTIAL SPECIFICATIONS FOR SYMMETRIC MPSOCS

Fabrizio Ferrandi, Luca Fossati, Marco Lattuada, Gianluca Palermo, Donatella Sciuto and Antonino Tumeo

Politecnico di Milano, Dipartimento di Elettronica e Informazione. Via Ponzio, 34/5, 20133 Milano, Italy. Ph: +39-02-2399-4009. Fax: +39-02-2399-3411[*]

{ferrandi,fossati,lattuada,gpalermo,sciuto,tumeo}@elet.polimi.it

Abstract: This paper presents an embedded system design toolchain for automatic generation of parallel code runnable on symmetric multiprocessor systems from an initial sequential specification written using the C language. We show how the initial C specification is translated in a modified system dependence graph with feedback edges (FSDG) composing the intermediate representation which is manipulated by the algorithm. Then we describe how this graph is partitioned and optimized: at the end of the process each partition (cluster of nodes) represents a different task. The parallel C code produced is such that the tasks can be dynamically scheduled on the target architecture; this is obtained thanks to the introduction of start conditions for each task. We present the experimental results obtained by applying our flow on the sequential code of the ADPCM and JPEG algorithms and by running the parallel specification, produced by the toolchain, on the target platform: with respect to the sequential specification, speedups up to 70% and 42% were obtained for the two bebchmarks respectively.

Keywords: Partitioning, Clustering, Automatic parallelization, thread decomposition, compilers for embedded systems, MPSoCs, FPGA.

1. INTRODUCTION

The technology trend continues to increase the computational power by enabling the incorporation of sophisticated functions in ever-smaller devices. However, power and heat dissipation, difficulties in increasing the clock frequency, and the need for technology reuse to reduce time-to-market push towards different solutions from the classic single-core or custom technology. A solution that is gaining widespread momentum consists of exploiting the

[*]Research partially funded by the European Community's Sixth Framework Programme, hArtes project.

Ferrandi, F., Fossati, L., Lattuada, M., Palermo, G., Sciuto, D., Tumeo, A, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 179–192.

inherent parallelism of applications, executing them on multiple off-the-shelf processor cores. Having separate cores on a single chip allows better usage of the chip surface, reduces wire-delay and dissipation problems and it provides more possibilities to exploit parallelism.

Unfortunately, the development of parallel applications is a complex task. Parallel programming is largely dependent on the availability of adequate software tools and environments and developers must contend with problems not encountered during sequential programming, namely: non-determinism, communication, synchronization, data partitioning and distribution, load-balancing, heterogeneity, shared or distributed memory, deadlocks, and race conditions.

This work tries to overcome some of these problems by proposing an approach for automatic parallelization of sequential programs. It focuses on a complete design flow, from the high level sequential C description of the application to its deployment on a multiprocessor system-on-chip prototype. In a first phase the sequential code is partitioned in tasks with a specific clustering algorithm that works on the System Dependence Graph (SDG). In a second phase the resulting task graph is optimized and the parallel C code is generated. The backend can produce OpenMP compliant code for functional validation on the host machine and C code that is runnable on multiprocessor system-on-chip architectures with a minimal operating system layer, as required by an embedded system. In both cases a mechanism to generate dynamically schedulable tasks has been defined. Through run-time evaluation of boolean conditions it is possible to determine the specific execution flow of the program and if a specific tasks needs spawning. Thanks to the mechanism implemented, tasks will be created as soon as only the real data dependences of the specific execution flow are satisfied, leading to an efficient exploitation of the underlying hardware.

The remainder of this paper is organized as follows: Section 2 gives an overview of the current state of the art on automatic parallelization techniques; Section 3 presents the target architecture used to execute the benchmarks. Section 4 introduces the flow we implemented focusing on the different steps which compose the partitioning and merging phases. Section 5 shows the numerical results obtained by the parallelization of the JPEG and ADPCM encoding algorithms and finally Section 5 concludes the paper.

2. RELATED WORK

MultiProcessor systems are becoming common, not only in the high performance segment, but also in the consumer and embedded ones. Developing programs for these new architectures is not easy: the developer needs to correctly decompose the application in order to enhance its performance and to exploit the multiple processing elements at his disposal. Sometimes, it is bet-

ter to rewrite a sequential description rather than to try porting it on complex, and often very different, architectures. Several strategies to ease the life of the developers, through partially automatization of the porting process, have been proposed; they are mainly related to two different approaches. The first one uses problem-solving environments which generate parallel programs starting from high level sequential descriptions. The other relies on machine independent code annotations. Our work adopts the first approach.

The parallelization process can be decomposed in several different steps. The initial specification needs to be parsed in an intermediate graph representation. This representation, which is used to explore the available parallelism, gets partitioned. After partitioning, an initial task graph is obtained. A task graph is a Directed Acyclic Graph (DAG) where each node describes a potential parallel code block. In a multiprocessor system it is necessary to allocate each task to one of the available processors. This allocation is usually realized through a two-step process: clustering and cluster-scheduling (merging). Many are the works related to the partitioning of the initial specification adopting specific intermediate representations. Among them, Girkar et al. [2] propose an intermediate representation, called Hierarchical Task Graph (HTG), which encapsulates minimal data and control dependence and which can be used for extraction of task level parallelism. Much of their work focuses on simplification of the conditions for execution of task nodes. Luis et al. [7] extend this work by using a Petri net model to represent parallel code, and they also apply some optimization techniques to minimize the overhead due to explicit synchronization.

Newburn and Shen [8], instead, present a complete flow for automatic parallelizazion through the PEDIGREE compiler. Their tool uses the Program Dependence Graphs (PDG) as intermediate representation and applies an heuristic to create overlapping inter-dependent threads. Their approach searches the PDG for control equivalent regions (i.e., groups of statements depending from the same control conditions) and then partition these region with a bottom up analysis. The resulting task graph is finally scheduled on subsets of processors of a shared memory multiprocessor architecture.

The clustering and merging phases have been widely discussed. Usually, these two phases are addressed separately. Well known deterministic clustering algorithms are dominant sequence clustering (DSC) by Yang and Gerasoulis [12], linear clustering by Kim and Browne [6] and Sarkar's internalization algorithm (SIA) [9]. On the other side, many researches explored the cluster-scheduling problem with evolutionary algorithms [4, 11]. An unified view is given by Kianzad and Bhattacharyya [5], who modified some of the deterministic clustering approaches introducing probability in the choice of elements for the clusters, and proposed an alternative single step evolutionary approach for both the clustering and cluster scheduling aspects.

Our approach starts from an intermediate representation which is not hierarchical like HTGs and PDGs, but instead flattens out all the dependence information at the same level. This creates bigger structures but gives the opportunity to extract more parallelism as it allows more complex explorations. Moreover, although our flow starts from the analysis of control-equivalent regions, we effectively partition them working on the data flow. This initial clustering is then optimized in order to create thread of homogeneous size, but we don't need any sort of static scheduling mechanism as we relie on dynamic start conditions.

3. TARGET ARCHITECTURE

The target architecture of the presented approach is a symmetric shared memory multiprocessor system-on-chip (MPSoC); we chose those systems because they are composed by many processing units, thus it is necessary to use applications composed by many concurrent tasks in order to exploit their processing power. We choose to test the produced code on a specific MPSoC prototype developed on Field Programmable Gate Array (FPGA), the CerberO architecture [10].

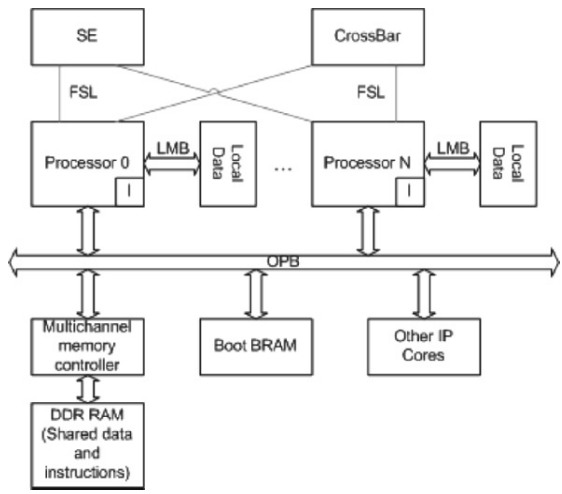


Figure 1. The target architecture of the presented approach

CerberO, shown in Figure 1, is realized connecting multiple Xilinx MicroBlaze softcores on a single shared Coreconnect On-Chip Peripheral Bus (OPB). Shared IPs and the controller for the shared external memory reside on the OPB bus. Each MicroBlaze in the system is connected to a local memory through the Local Memory Busses (LMB) for private data. The shared instructions and data segments of the applications reside on the (slower) external DDR mem-

ory, while private data are saved in the local memories. The addressing space of each processor is partitioned in two parts: a private part and a shared part. The private space is accessible only by the local processor, while the shared one is equally seen by all components of the system. Instructions are cached by each processor, while data are explicitly moved from the shared to the fast, low latency private memory.

Synchronization mechanisms are provided through a dedicated module, the Synchronization Engine (SE), connected to each processor through the Fast Simplex Link (FSL) point to point connections. The SE is a configurable, centralized hardware lock/barrier manager that permits atomic accesses to shared memory locations and shared peripherals.

On top of the CerberO architecture, we developed a thin operating system (OS) layer that permits to dynamically schedule and allocate threads.

4. PARALLELIZATION

This section introduces our flow to automatically produce multi-threaded programs using as input a sequential specification written in the C language. The toolchain manages the partitioning of the sequential description and generates code that can be dynamically scheduled on the target architecture using specific boolean task start conditions.

Figure 2 provides an overview of the entire flow. The sequential C code is compiled with a slightly modified version of the *GNU C Compiler* (GCC) version 4.0 and the internal structures generated by the compiler are dumped. From these structures, our tool suite *PandA* creates an abstract representation of the program in terms of several graphs describing data and control dependence. The C to C partitioning algorithm works on a modified system dependence graph (SDG). We define the SDG as a graph in which both data dependence and control dependence are represented for each procedure. This graph gets clustered and optimized, and the resulting task graph is then converted back in parallel C code by the specific backend. This parallel C code can finally be compiled for the target MPSoC architecture.

FSDG Creation

In this phase of the process *PandA* parses the dump of the intermediate representation of the modified *GCC* 4.0 compiler and creates a data structure containing all the relevant information expressed in the initial specification. From the version 3.5/4.0 of the GCC compiler the front-ends parse the source language producing GENERIC trees, which are then turned into GIMPLE. GENERIC and GIMPLE are language independent, tree based representations of the source specification [1]. Although GIMPLE has no control flow struc-

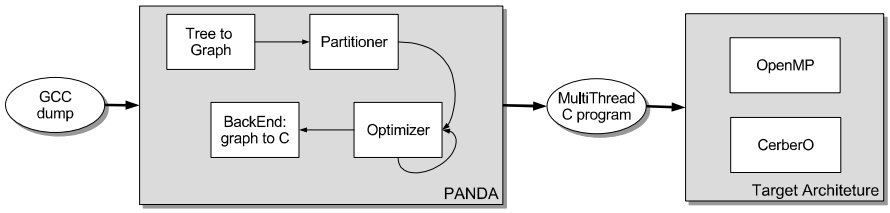


Figure 2. Overview of the toolchain

tures, GCC also builds the control flow graph (CFG) to perform language independent optimizations.

The GCC analysis and the GIMPLE parsing correspond to the first step performed by the PandA framework to analyze the input sequential program; several graph structures describing control and data dependencies in the program are then derived. Each graph is a different view of the dependencies among the operations of the initial specification.

In particular, the proposed partitioning algorithm executes a post-processing analysis on the dependencies represented by a SDG [3] extended by introducting feedback edges (FSDG). A sample FSDG is shown in Figure 3. Vertices are statements (round and square boxes) or predicate expressions (rhombus boxes). Grey solid edges represent data dependencies and describe flow of data between statements or expressions. Black edges represent control dependence, and express control conditions on which the execution of a statement or expression depends. Black dashed edges represent both control and data dependencies. Grey dashed edges represent feedback edges, which makes possible to distinguish nodes belonging to loops from other nodes. Finally, the entry node represents the entry to the procedure. It is worth noting that all the loops are converted in do-while loops, since this code transformation simplifies the management of the exit condition. This graph also allows the recognition of control-equivalent regions, which are groups of nodes that descend from the same condition (True or False) of a father predicate node.

Partitioning algorithm

The partitioning phase uses the FSDG as defined above as its input. The first step of the algorithm analyses feedback edges and generates partitions of nodes from the initial FSDG: one for each loop and one grouping all the nodes not in loops. Thanks to this procedure, parallelism can be extracted also for

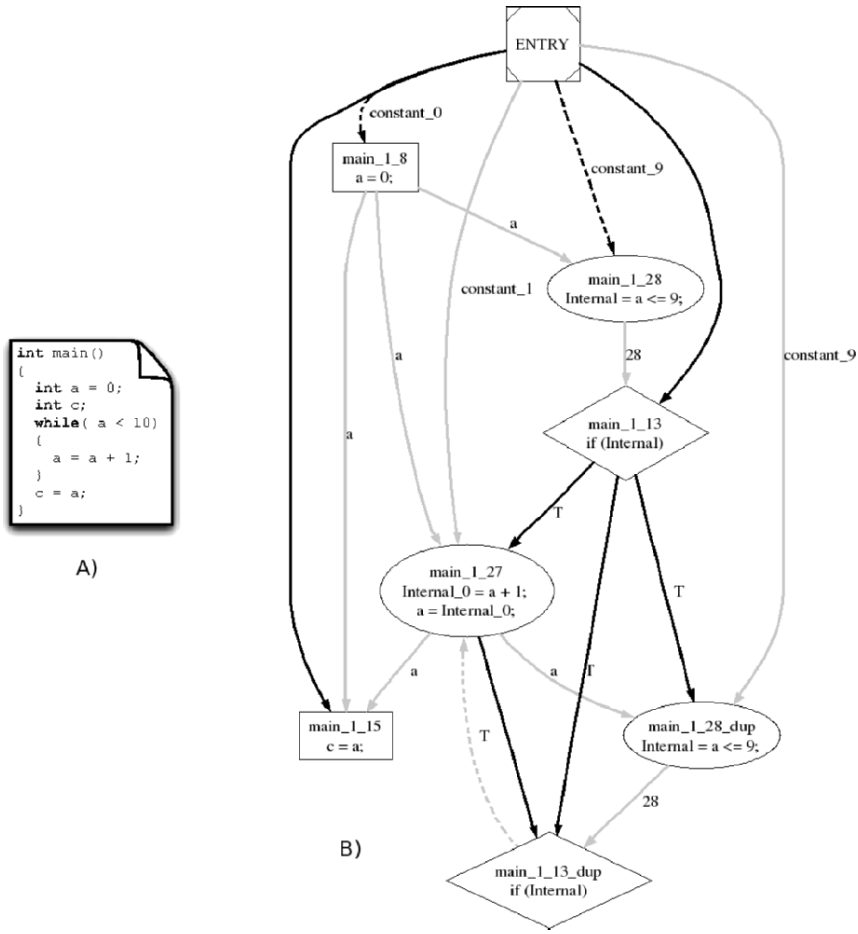


Figure 3. Example of an FSDG graph

the code inside the loops, instead of considering loops as atomic entities. The following steps of the algorithm consider these partitions separately, one at a time, and try to extract only the parallelism contained in each of them.

After identifying the loop partitions, the algorithm performs an analysis of the control edges in order to recognize the *control-equivalent* regions. Statement nodes descending from the same branch condition (True or False) of a predicate node are grouped together, forming a single control-equivalent region, as in Figure 4, A. The procedure runs as long as all the predicate nodes are analysed.

Since the nodes in each control-equivalent region can only be data dependent among each other, each region is a potential candidate to form a parallel

task. For each control-equivalent region data dependence analysis is then executed, grouping together nodes that are inter-dependent and, hence, must be serially executed, like in Figure 4, B. The analysis starts from a generic node in a control-equivalent region with a depth-first exploration. A node is added to the cluster being formed if it is dependent from one and only one node or if it is dependent from more than one node, but all its predecessors have already been added to the current cluster. Otherwise, the cluster is closed and the generation of a new set starts. These operations are iterated until all the nodes in the control-equivalent partition are added to a set.

The final result of this procedure is a clustered FSDG graph, which represents a potential task graph, Figure 4, C. Each partition of the clustered graph represents a single basic block of instructions, with none, or minimal interdependence. Clusters that not depend on each others represent blocks of code that can potentially execute in parallel. Edges among clusters express data dependences among blocks of code, thus the data represented by in-edges of a partition must be ready before the code in that partition can start.

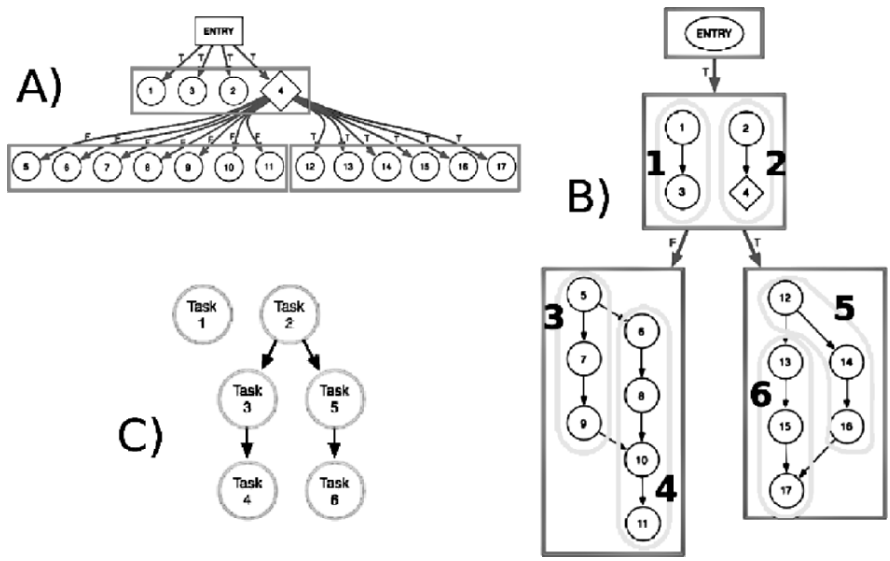


Figure 4. Example of clustering: A) Control equivalent regions gets analyzed, B) Control Equivalent Regions gets partitioned, C) Corresponding Task Graph

Optimizations

Even if the prototyping platform has just a small operating system layer with little overhead due to thread management, shared memories must be atomically accessed in order to create the task structures and verify which processing elements are free. The partitioning phase, explained in Section 3, tends to produce too many small clusters, with the effect that the overhead of task management could be higher than the advantages given by concurrent execution. Thus, we decided to introduce an optimization phase aimed at grouping clusters together. Two different techniques are used: optimizations based on control dependencies and optimizations based on data dependencies.

Control structures, such as the `if` clause, must be executed before the code situated beneath them, otherwise we would have speculation: it seems, then, reasonable to put the control statement and the instructions which depend on it in the same task, in particular if there are no other nested control clauses. Another optimization consists in grouping the `then` and `else` clauses in the same cluster: they are mutually exclusive, the parallelism is not increased if they are in separate clusters. If a control clause does not get optimized, as it could happen if there are many nested control statements, it is replicated in each one of the clusters containing control dependent instructions. This means that the only type of dependencies remaining between different clusters are *data dependencies*, making the subsequent management of the scheduling easier.

Data dependent clusters can be joined together to form a bigger cluster; the candidates for joining are those clusters containing a number of nodes (instructions) smaller than n; this number roughly represents the overhead due to the management of a task. When a small cluster is encountered, the algorithm tries to join it with the successor (say that a task b is the successor of a if b has a control or data dependence on a); this operation is successfully carried out when all the data dependences on edges exiting from a have the same target cluster b. These steps are repeated until no more clusters are joined or no more clusters smaller than n exist.

Task creation

This part of the partitioning flow is used to translate the final clustered FSDG into specific data structures effectively representing the tasks, their input/output parameters and the relationships among them. The first step consists in the identification of the task variables: the edges coming from the special node `ENTRY` are associated with global variables; the edges entering in a cluster represent the input parameters of the task, while the outgoing edges the output

parameters. Finally, the edges whose both source and destination nodes are contained in the same cluster form the local variables of the task. Note that, of course, it may happen that two different edges are associated with the same variable: in this case just one variable is instantiated.

The next operation consists in the computation of the start conditions for each task; these conditions enable *dynamic scheduling* of the tasks. Through dynamic scheduling the application can decide at runtime if a task should start or not. Consider, for instance, that the application needs to start a task such as the one in Figure 5. The arrows pointing out of each variable indicate that they use a value produced by other tasks. To safely start the new task the application should wait that `a`, `b` and `c` are written by the preceding tasks. Hence the tool chain would have to determine a static scheduling and organize the execution flow in order to guarantee that all the preceding tasks have correctly written their values to cover for each possible execution path. Actually not all three variables are used together, using `a` and `b` implies that `c` is not used and vice-versa. A static scheduling would limit the exploitable parallelism, since the new tasks could not be overlapped either to the tasks producing `a` and `b` or to the task producing `c`. However, when the value of the condition `C1` is determined, it is decided which branch is going to be executed: in case the *else* branch is taken, the new tasks only need to wait for the task which produces `c`, otherwise for the tasks which produce `a` and `b` (this, of course, works if `a`, `b` and `c` are produced by different tasks). So, if we allow the application to precompute the value of the condition, it could fully take advantage of the support for dynamic scheduling of the target architecture and, effectively, start the new task as soon as only the true data dependencies of the specific execution flow are satisfied.

Taking these considerations into account, we enabled our Task Creator to generate, for each task, a specific start condition related to the runtime execution path of the application. A valid start condition for a task has to satisfy the following points: (1) a task must be started only once (i.e. only one of the predecessors of the task can start it) and (2) all the parameters necessary for a correct execution of the task must have already been computed when the condition evaluates to true. To enforce the first point we use a simple boolean variable which is set to true when the task starts; the second point is much more complicated since, depending on the execution path which will be taken, different variables are used.

To generate the start condition, the algorithm initially explores all the input parameters belonging to the task to be started. Parameters used in the same control region (i.e. all in the true or false branch) are put in an `and` condition (all of them must be ready if that control region is going to be executed). All the resulting "and" expressions (one for each control region) are joined by an "or" operator.

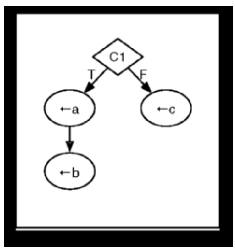


Figure 5. A task, depending on the particular execution flow, may need different parameters

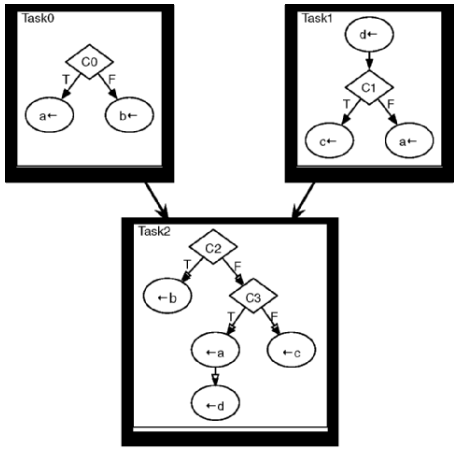


Figure 6. Typical sequence of tasks: depending on the value of *C2* and *C3*, Task2 uses different parameters

The second step consists in the exploration of the preceding tasks, the ones which have an edge entering in the current task, looking for the locations where the parameters, input to the task to start, are written. In case there are more control flows from which a parameter is written, all the corresponding paths are joined in an "or" expression. Using the diagram in Figure 6 as example, we would have the condition for *Task2*:

$$C2 \cdot \gamma\,(b) + \neg C2 \cdot [C3 \cdot (\gamma\,(a) \cdot \gamma\,(d)) + C3 \cdot \gamma\,(c)] \tag{1}$$

$\gamma\,(x)$ identifies all the possible paths, in the preceding tasks, which compute x; in the example: $\gamma\,(a) = C0 \cdot T0 + \neg C1 \cdot T1$, where $T0$ and $T1$ are boolean variables equal to true if Task0 and Task1 has already ended. After computing all the necessary $\gamma\,(x)$ functions we need to complete the start condition by indicating that Task2 can start only if it has not started yet, so we put Equation 1 in "and" with $\neg T2_s$ which is true if Task2 has already started. Since the resulting condition is usually long, but contains many redundant elements, we use BDDs (Binary Decision Diagrams) to reduce its complexity. The condition is inserted at the end of both Task0 and Task1: the one which ends last will evaluate the condition to `true` and it will actually launch the execution of Task2.

The last phase of the flow is the generation of the final parallel C code starting from the representation of the tasks created in the previous steps. This work is done by a specific C Writer backend, that needs to produce code in a syntax compliant with the primitives supported by the target architecture. The

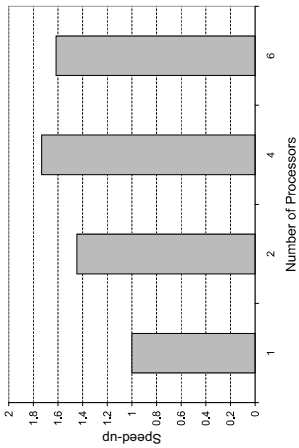


Figure 7. ADPCM performance

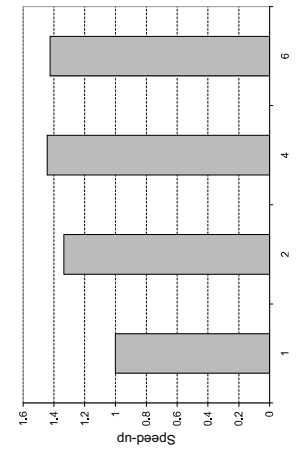


Figure 8. JPEG performance

backend can produce both OpenMP compliant code for functional validation and code runnable on the target platform for final implementation.

5. EXPERIMENTAL EVALUATION

The effectiveness of our partitioning flow was verified running the parallelized programs on the CerberO architecture. Execution time on different runs, using a different number of processors was measured. When analyzing the results, it must be taken into consideration that the execution time is affected not only by the degree of parallelism extracted, but also by the overhead introduced by the thread management routines of the underlying OS layer. For this reason, in order to perform a fair comparison, the original sequential program was slightly modified in order to execute it on top of the OS layer.

Figure 7 shows the speedup obtainable by running the ADPCM algorithm on a different number of processors. The maximum speedup is obtained when four processors are used, with a performance roughly 70% higher than the results on the single processor platform. With more than four processors the execution speed starts lowering. This behavior is due to the fact that the parallelized program contains at most four threads which have no inter-dependences and can, hence, run in parallel; using more processors does not increase the parallelism, but it does increase the synchronization overhead. The speed ups obtained are similar to the average results shown in [8] but, while its authors exploit also fine grained ILP parallelism, our target architecture adopts much simpler processor cores.

Figure 8 represents the speedup obtained by parallelizing part of the JPEG algorithm. The most computationally intensive kernels of the algorithm, the

RGB-to-YUV color space conversion and the 2D-DCT, have been parallelized. The RBG-to-YUV and 2D-DCT algorithms account for the 70% of the execution time of the sequential code on our target platform, while the remaining 30% comprises the reading of the input and the writing of the output image which are not parallelizable. As for the ADPCM algorithm, the maximum parallelism extracted by our tool chain is four, so using more than four processors leads to performance degradation for unnecessary synchronization operations and more load on the shared bus and memories of the target architecture. For the whole JPEG algorithm the maximum speedup reached is 42%.

6. CONCLUDING REMARKS

This paper presented our design flow for automatic parallelization of a sequential specification targeting a homogeneous MPSoC prototype. The main contributions of this paper can be summarized as follows: (1) it proposes a complete design flow from sequential C to parallel code runnable on homogeneous multiprocessor systems, (2) it describes a partitioning algorithm to generate parallel code transforming all control dependencies in data dependencies from sequential C and, finally, (3) it introduces a dynamic task scheduling model with specific start conditions to spawn the parallel threads of the application, without requiring complex operating system support by the target architecture. The flow has been applied to several standard applications, and the results obtained with the ADPCM and the JPEG algorithms on a MPSoC prototype on FPGA show promising levels of parallelism extracted, with speedups up to 70% and 42% respectively.

REFERENCES

[1] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[2] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–178, March 1992.

[3] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39(4):229–243, 2004.

[4] E.S.H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5:113–120, 1994.

[5] V. Kianzad and S.S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(17):667–680, July 2006.

[6] S.J. Kim and J.C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Int. Conference on Parallel Processing*, pages 1–8, 1988.

[7] J.P. Luis, C.G. Carvalho, and J.C. Delgado. Parallelism extraction in acyclic code. In *Parallel and Distributed Processing, 1996. PDP '96. Proceedings of the Fourth Euromicro Workshop on*, pages 437–447, Braga, January 1996.

[8] C.J. Newburn and J.P. Shen. Automatic partitioning of signal processing programs for symmetric multiprocessors. In *PACT '96*, 1996.

[9] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[10] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. A design kit for a fully working shared memory multiprocessor on FPGA. In *Proceedings of ACM GLSVLSI'07 – Great Lakes Symposium on VLSI*, March 11-13 2007.

[11] W.K. Wang. Process scheduling using genetic algorithms. In *IEEE Symposium on Parallel and Distributed Processing*, pages 638–641, 1995.

[12] T. Yang and A.Gerasoulis. Dsc: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1994.

AN INTERACTIVE MODEL RE-CODER FOR EFFICIENT SOC SPECIFICATION*

Pramod Chandraiah and Rainer Dömer
Center for Embedded Computer Systems
University of California Irvine
pramodc@uci.edu, doemer@uci.edu

Abstract To overcome the complexity in System-on-Chip (SoC) design, researchers have developed sophisticated design flows that significantly reduce the development time through automation. However, while much work has focused on synthesis and exploration tools, little has been done to support the designer in writing and rewriting SoC models. In fact, our studies on industrial size examples have shown that about 90% of the system design time is spent on coding and re-coding of SLDL models, even in the presence of algorithms given in the form of C code. Since the quality of the design model has tremendous impact on the cost and quality of the resulting system implementation, creating and optimizing the model is a critical task toward successful SoC design.
In this paper, we present an interactive source re-coder which integrates static analysis and code transformation tools into an editor to assist the designer in tedious modeling and optimization tasks. This novel approach allows the designer to use her/his limited modeling time efficiently, and thus yields significant gains in productivity.

Keywords: System-level Design, Specification Modeling, Embedded systems, System-on-Chip

1. INTRODUCTION

In the past, the system-level design community has focused on solving various problems of system synthesis. Researchers have been working towards design automation at various abstraction levels with the goal to automate steps in the design process and reduce the design time. Motivated by the need to meet the time to market and aggressive design goals like low power, high performance, and low cost, researchers have

*This work was supported in part by Nicholas Endowment through the Henry T. Nicholas III Research Fellowship.

Chandraiah, P., Dömer, R, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 193–206.

proposed various design methodologies for effective design development, including top-down and bottom-up approaches. All these technological advances have significantly reduced the development time of embedded systems. However, design time is still a bottleneck in the production of systems, and further reduction through automation is necessary. One critical aspect neglected in optimization efforts so far is the design specification phase, where the intended design is captured and modeled for use in the design flow.
Each design methodology expects a specific type of input model and most methodologies depend on intermediate design models for interaction between tools and the designer. The specification needs to be either hand-written from scratch, or modified from a reference model. While much of the research has focused on SoC synthesis and refinement tools, little has been done to support the designer in forming these models.

1.1 Motivation

In order to study the intricacies and complications involved in writing a system specification, we have applied a top-down design methodology, as shown in Figure 1, to the example of a multimedia application, a MP3 audio decoder. Here, the design process starts with an abstract specification model which is then refined to create models at lower abstraction levels, including transaction-level, bus-functional and implementation models. After a series of refinement steps, an actual implementation model is finally derived. Each of the refinement steps in the design flow is automated to the extent that model generation is fully automatic, and the designer has to only make the design decisions such as component allocation, mapping and scheduling. Due to this automation, we were able to implement our MP3 decoder model, an industry-size application, in less than a week [1]. In contrast, manually re-coding the reference implementation into a specification model took 12-14 weeks. Writing and re-writing this model was

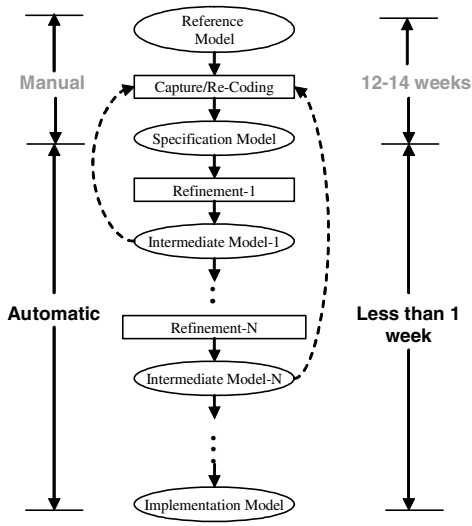


Figure 1. Motivation: Design time of MP3 decoder in a refinement-based design flow.

the main bottleneck of the whole process. More than 90% of the overall design time was spent in creating the specification model.
Also, we need to emphasize that specification capturing is not a one time task. Every time a change in the design is required for a successful refinement step, it is necessary to re-code/change the input specification (as shown by back arrows in Figure 1), making the whole task of specification writing iterative. Such interruptions in the design flow cause costly delays. The problem of lengthy re-coding of models is not a problem specific to a top-down methodology. Its importance is also emphasized in [12, 6].
In conclusion, any step towards automation of model coding and re-coding is highly desirable and will likely improve overall design time significantly.
In Section 2, we discuss model re-coding and related work. In Section 3, we present our solution to the modeling problem, an interactive source re-coder. Section 4 lists our experimental results. Finally, we will draw conclusions and outline future work in Section 5.

2. MODEL RE-CODING

Reference models of the algorithm obtained from software vendors, standardizing committees (eg. ISO/IEC) or similar sources act as a good starting point for creating the SoC specification. Depending on the design flow, these reference models must be recoded in System Level Description Languages (SLDLs) such as SystemC [7], SpecC [4] or SystemVerilog [18]. Apart from the recoding of the C model into a SLDL model, various modeling guidelines recommended by the design flow must also be incorporated into the model. Typical modeling guidelines [5] include, clear separation of communication and computation, sufficient computational granularity, and exposing concurrency.

2.1 Automated Re-Coding

Apart from the time consuming textual operations, re-coding a system model involves a lot of decision making. Many of these decisions can only be taken by the designer. For example, if the designer decides to map a C function onto a separate hardware component, the model needs to be re-coded to encapsulate this function in a separate block (behavior/module). In the absence of efficient hardware/software partitioning tools, this decision needs to be taken by the designer. However, the tedious recoding of the model can be automated.
To leverage the idea of re-coding automation, we need to distinguish re-coding tasks that can be automated from decision tasks that require

designer's experience. Textual re-coding operations such as introducing blocks, changing scope of variables, grouping functions, etc. can be performed automatically, if the decision is made by the designer. By such automation, the designer is relieved from mundane text editing tasks and can focus on actual modeling decisions. To address such issues, it is efficient to have a re-coder that is interactive and applies the changes on the fly. Since the model being derived is under full control of the designer, the output of such a tool can suit many design flows based on similar C-like languages.

2.2 Related Work

In this section, we will briefly present some work related to re-coding. First, we will look at the general area of program transformations and then focus on interactive ones.

2.2.1 Program transformations. In the past, researchers have developed program transformations for many different areas, including to improve aesthetics, to parallelize applications, and to perform high level synthesis (HLS). For example, the SUIF compiler [9] identifies loop level parallelism in a program and transforms a sequential program into a single-program, multiple data program. The Spark HLS framework [8] applies source and low level parallelizing transformations to a design to improve the quality of the target hardware. Unlike SUIF and Spark, our re-coder specifically aims at re-coding a C reference implementation into models in SLDLs suitable for design space exploration and system synthesis. The SpecSyn [3] synthesis system expects input specification in the SpecCharts language, but provides no facilities to create the initial SpecChart model. Our transformations are interactive and give the designer complete control ("designer-in-the-loop") to code/re-code the C reference model in order to arrive at the most suitable design implementation.

2.2.2 Interactive program transformations. To compare existing interactive coding environments and their capabilities, we distinguish textual abilities, syntax awareness, semantics awareness, analysis and program transformation capabilities. Figure 2 shows a set of tools and the extent to which they meet these capabilities. A simple *text-aware* editor, such as Textpad, supports plain textual entry and basic formatting. Better editors are syntax aware. By *syntax awareness* we mean the ability to recognize syntactical elements of the language. For example, highlighting of keywords and matching of braces requires syntax awareness. A *semantics- aware* editor provides advanced features

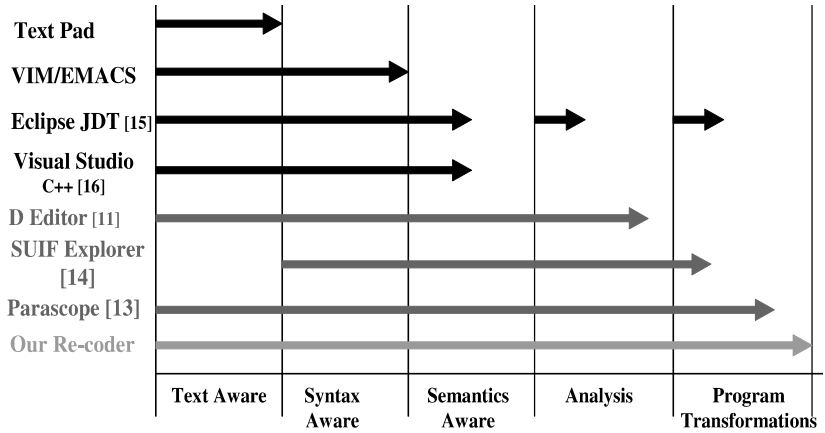


Figure 2. Capabilities of Interactive coding environments.

like context-assist, auto-complete, error indications, etc. Examples include Eclipse Java Development Tool (JDT) [15], and Microsoft Visual studio [16]. In addition to syntax and semantics, the Fortran D editor [11] is equipped to provide dependency analysis and other information about parallelism and communication.

The ParaScope editor [13] for Fortran and SUIF explorer [14] for C/Fortran provide program transformations to parallelize a program and relieve the programmer from tedious manual typing. SUIF explorer provides graphical means of setting compiler directives, but does not support editing. ParaScope provides the user with powerful interactive program transformations and reconstructs the dependency information, incrementally, while editing. Of all, ParaScope combines the most features. Our goal is to build similar and more advanced capabilities into our source re-coder, aiming specifically at analysis and transformations necessary to re-code a C reference source into a SLDL model. It is also necessary to mention that our transformations are generic and will not address specific transformations such as reimplementing data structures, replacing a slow algorithm with a faster implementation, and so on. However, such complex transformations can be realized by the designer using the set of generic transformations provided by our re-coder.

3. INTERACTIVE SOURCE RE-CODER

To aid the designer in coding and re-coding, we propose a source *re-coder*. Our source re-coder is a controlled, interactive approach to implement analysis and refinement tasks. In other words, it is an intelligent union of editor, compiler, and powerful transformation and analysis tools.

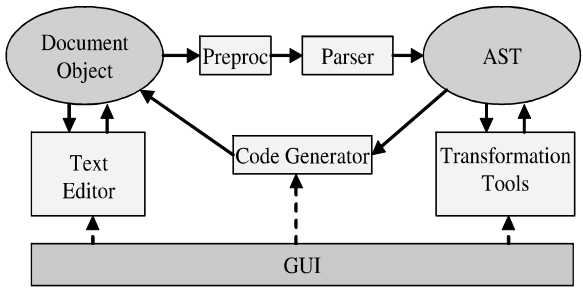


Figure 3. Conceptual Structure of the Source Re-Coder.

Unlike other program transformation tools, our re-coder keeps the designer in the loop and provides complete control to generate and modify a model suitable for her/his design flow. By making the re-coding process interactive, we rely on the designer to concur, augment or overrule the analysis results of the tool, and use the combined intelligence of the designer and the re-coder for the modeling tasks. Our re-coder supports re-modeling of SLDL models at all levels of abstraction. It can be used to re-code intermediate design models as well as the reference C implementation to generate the initial specification model. The conceptual structure of our source re-coder is shown in Figure 3. It consists of 5 main components:

- Textual editor maintaining the textual document object
- Abstract Syntax Tree (AST) of the design model
- Preprocessor and Parser to convert the document object into AST
- Transformation and analysis tool set
- Code generator to apply changes in the AST to the document object

3.1 Editor

We have chosen a QT [19] and Scintilla [17] based textual editor as the front-end of our source re-coder. The basic document object is based on the data structures in the Andrew text editor [10]. This editor has built-in support for features like syntax highlighting, auto-completion, search, ctags, text folding, bookmarks, undo-redo, and more, for programming languages including C and C++. As an initial step, we have adapted and extended these features for support of SystemC and SpecC SLDLs. The designer makes all the re-coding decisions through the Graphical User Interface (GUI) provided by the editor, for example, through extended context-menus.

3.2 Abstract Syntax Tree

For the editor, the source code being edited is a mere text. In order to analyze and transform the code, the source re-coder needs to recognize the complete structure of the program. When the source code of the design is loaded from a file, a scanner/parser is invoked which creates an object-oriented data structure, AST [21]. internal representation. Figure 4 gives an overview of the amount and the kind of information embedded in the AST object-oriented data structure. The AST preserves all structural information, block, channel, port, and interface, along with C constructs and file information. The AST also provides a set of operations on each object. The completeness of the AST makes the correspondence between the data structure and the text in the editor possible. Locating an object in the AST for a given line and column in the source window, and vice-versa, is achieved by maintaining line and column information in the AST.

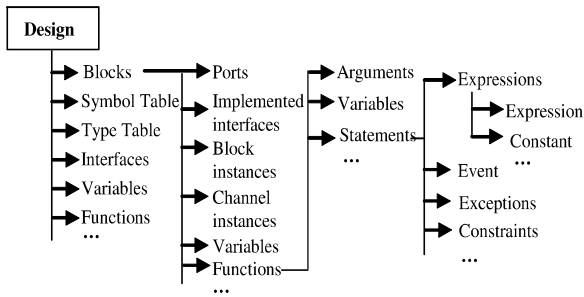


Figure 4. Information in the Abstract Syntax Tree.

3.3 Preprocessor and Parser

When the design is initially loaded, the preprocessor is run on the source to process file inclusions and macros. The lexer and the parser are extended to provide advanced syntax highlighting in the editor. In addition, the parser creates the AST.

Since we use two separate data structures to maintain text data and syntax tree, we need to synchronize between the two. When the designer modifies the text in the document object, these changes need to be reflected in the AST. Our parser provides this synchronization by applying the changes to the AST immediately while editing [1].

[1]If the text added results in an invalid source code (syntax error), which naturally happens when the designer is typing in the code, the parser parses the program until the point of error and creates only a partial AST. Transformations cannot be invoked during such transient phases, but as soon as the typing is complete and the code is parse'able again, the AST is updated immediately and transformations are available again.

3.4 Analysis and Transformation Tools

The transformation and analysis tool set is the heart of our source re-coder. All re-coding tasks invoked by the user are implemented by these refinement tools. When the designer points to an object in the source window, a node corresponding to the pointed co-ordinates is located in the AST, and a list of available and possible tasks are provided in a context menu. We categorize our re-coding operations into 3 classes.

Structural transformations change the structure of the program by introducing/removing computational blocks, channels and functions. In this category, we further distinguish

- Granulating transformations
- Composing transformations
- Re-organizing transformations

Functional transformations modify computational blocks and functions. For example, the designer may change the interface of blocks by introducing or removing ports. This category is further subdivided into:

- Transformations to contain communication
- Transformation to break dependencies
- Pruning transformations

Analysis functions provide dependency information of an object without introducing any changes to it. Analysis functions also provide information to the designer about potential parallelism at function and block level. In addition, analysis to deal with pointers is also provided.

The description of these transformations is beyond the scope of this paper.

3.5 Code Generator

Any modification in the AST introduced by the transformation tools needs to be reflected in the text in the editor. This synchronization is provided by the code generator, which generates SLDL source code from the modified AST. As such, the code generator is the corresponding tool to the preprocessor and parser which update the AST for text changes.

Table 1. Time to create AST [secs]

Operation	**Simple**	**JPEG**	**MP3**	**GSM**
Lines of code	**174**	**1642**	**7086**	**7492**
Size in bytes	**2757**	**32863**	**223464**	**217414**
Read file, create doc.obj.	0.099	0.104	0.220	0.208
Preprocess	0.027 (0.001)	0.026 (0.001)	0.012 (0.014)	0.028 (0.013)
Parse/Build AST	0.023 (0.034)	0.055 (0.046)	0.431 (0.151)	0.299 (0.147)
Update editor view	0.005	0.005	0.011	0.010
Total time	0.155 (0.035)	0.190 (0.047)	0.675 (0.165)	0.546 (0.160)

4. EXPERIMENTS AND RESULTS

To demonstrate the feasibility and benefits of our re-coding approach, we have implemented the software infrastructure of the proposed re-coder and some of the analysis and transformations to recode a C-based SLDL model. Figure 5 shows the source re-coder at run time. Implemented transformations include

- Localizing global variables
- Re-scoping variables across structural hierarchy
- Synchronizing variable access by introducing channels
- Introducing/Deleting ports in behaviors

Besides, other basic transformations such as semantic-sensitive renaming, deletion, and dependency-analysis functions are also incorporated into the recoder The details of the above transformations are discussed in [2].
To prove the concept of instant refinement in the editor window, we will focus on the responsiveness of the recoder. Following this, we quantify the productivity gains achieved by using our source re-coder.

4.1 Results for Responsiveness

One concern with any interactive environment is its responsiveness. Since our re-coder, in addition to textual editing, builds and maintains a complete AST, and performs complex analysis and transformations, we have measured the time consumed by various operations. Table 1 shows the time it takes to load and build a design from a file. For 4 different designs of varying sizes, the table summarizes timing results for

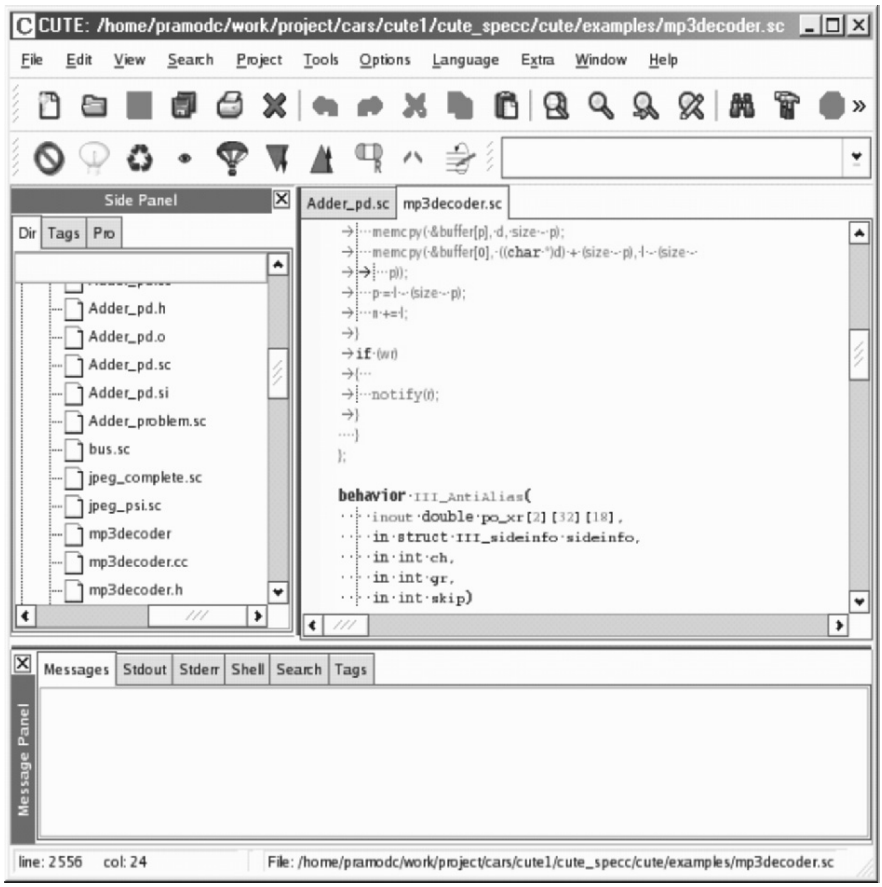


Figure 5. Screen shot of Source Re-Coder.

each sub-operation. The measurements were performed on a Pentium-4 3GHz Linux PC. Even for the designs with more than 7000 lines of code, the responsiveness of the re-coder is satisfactory. Clearly, the creation of the editor data structure and the AST are the most time intensive operations. In the current implementation of our re-coder, synchronization between the editor document object and the AST are implemented using file I/O. These file I/O overheads are indicated in parenthesis. Clearly, our re-coder is sufficiently responsive, even in the presence of the file I/O overhead. In the future, if the performance of the source re-coder becomes critical, some of the overhead will be eliminated. We will also consider incremental updates of the data structures, which promise to cut down the processing times even further.

Table 2 shows the timing of different transformations as experienced by the designer while using the source re-coder on the MP3 design example.

Table 2. Re-coding time in MP3 example[secs] [2]

Operation	**No. of times applied**	**Lines changed**	**Interactive Re-coding time (secs)**
Localizing variables	26	330	90
Re-scope	38	839	126
Synchronize with channels	6	172	30
Total	70	1341	246

Table 3. Productivity gains [2]

Transformations	**JPEG**	**MP3**	**GSM**
Global Variables localized	8	70	83
New Ports added	2	146	163
New Channels added	1	6	2
Re-coding time (secs)	27	246	260
Estimated Manual time (mins)	53	497	585
Productivity factor	117	121	135

The table lists 3 transformations used to create explicit communication structure [2] in the SoC models, the number of times they were invoked for the MP3 example, and the resulting number of lines affected. The interactive recoding time in the table is the wall clock time as experienced by the designer. These otherwise time consuming transformations can be implemented in seconds.

4.2 Productivity gain

We will now assess the productivity gains resulting from our source re-coder. We have applied the above mentioned transformations on different design examples and compared the design time taken to implement these manually over using our source recoder. Table 3 shows the productivity gain for different transformations. The manual time is obtained by actually realizing the transformations manually for a set of 10 variables and extrapolating the results for all the variables. The manual editing of the code was performed using Vim [20], an advanced text editor with block editing capability. Re-coding time is the time taken to implement the same transformations using the source re-coder by the same designer. Given the decision information, our source re-coder needs less than a second to implement these transformations. On the contrary, it will take fractions of an hour (instead of fraction of a second) to manually implement these transformations.
In general, measuring productivity is a difficult task. Factors such as designer's experience and tools used must be considered for accurate

measurement of productivity gains. However, in our experiments, since productivity gains are in the order of hundreds, any increased accuracy in the measurements will not significantly influence the end conclusions. In other words, since our improvements are by multiple orders of magnitudes, small adjustments to measurement accuracy will not make any significant difference.

5. SUMMARY AND CONCLUSIONS

In model-based design flows using automated synthesis tools, the creation and maintenance of the design model is often the main bottleneck towards further reducing of the design time. Little or no tool support is typically available for specification coding and re-coding. Usually, designers have to edit the design model manually, using simple text-based editors, requiring significant effort and time in tedious coding. At the same time, modeling is a critical task in SoC design, as the quality of the resulting implementation directly depends on the quality of the input model.

In this paper, we have proposed a novel approach to design specification and modeling, that is based on interactive decision making by the designer ("designer-in-the-loop"), and automation of model analysis and transformation tasks. Eliminating mundane and error-prone text editing tasks, our recoding approach utilizes the precious time and effort of the system designer efficiently.

In particular, we have introduced an interactive source re-coder which integrates compilation, analysis and transformation tools into a text-based editor, to assist the designer in modeling and re-modeling of SoC designs. Our source re-coder is fully text-, syntax-, and semantics-aware, enabling powerful model analysis and transformation operations.

Our approach is especially useful in re-coding of reference models into SoC specification models, which is often the case for new designs. Our source re-coder aids the designer in the analysis and comprehension of the reference model, as well as in restructuring of the model towards a well-specified input model for the system design flow.

For initial analysis and transformation tasks, our experimental results clearly demonstrate that an interactive approach is not only feasible, but also effective. Analysis results or transformed code are presented to the user instantaneously, relieving the designer from tedious coding. Moreover, we have demonstrated tremendous productivity gains through the reduction of modeling time.

For the future, we will extend our approach to include further analysis and transformation tasks, including coupling it with system profiling and estimation tools.

References

[1] P. Chandraiah and R. Dömer. Specification and design of an mp3 audio decoder. Technical report, Center for Embedded Computer Systems, May 2005.

[2] P. Chandraiah, J. Peng, and R. Dömer. Creating explicit communication in soc models using interactive re-coding. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2007.

[3] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[4] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[5] A. Gerstlauer, R. Dömer, J. Peng, and D. D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[6] A. Gerstlauer, S. Zhao, D. D. Gajski, and A. M. Horak. SpecC system-level design methodology applied to the design of a GSM vocoder. In *Proceedings of the Workshop of Synthesis and System Integration of Mixed Information Technologies*, Kyoto, Japan, April 2000.

[7] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[8] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):441–470, 2004.

[9] M. W. Hall, J.-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.

[10] W. J. Hansen. Data structures in a bit-mapped text editor. *Byte Magazine*, 12(1):183–189, January 1987.

[11] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. K. Warren. The D editor: a new interactive parallel programming tool. In *Supercomputing*, 1994.

[12] A. Jerraya, H. Tenhunen, and W. Wolf. Guest editors' introduction: Multiprocessor systems-on-chips. *Computer*, 38(7):36–40, 2005.

[13] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the ParaScope Editor. In *ACM International Conference on Supercomputing*, Cologne, Germany, 1991.

[14] S.-W. Liao, A. Diwan, R. P. B. Jr., A. M. Ghuloum, and M. S. Lam. SUIF explorer: An interactive and interprocedural parallelizer. In *Principles Practice of Parallel Programming*, 1999.

[15] Eclipse java development tool-kit. http://eclipse.org/jdt/index.html.

[16] Microsoft visual studio. http://msdn.microsoft.com/vstudio/.

[17] Scintilla source code editing component. http://www.scintilla.org.

[18] S. Sutherland, S. Davidmann, P. Flake, and P. Moorby. *System Verilog for Design: A Guide to Using System Verilog for Hardware Design and Modeling*. Kluwer Academic Publishers, 2004.

[19] Trolltech Inc. Qt application development framework. http://www.trolltech.com/products/qt/.

[20] Vim, advanced text editor. http://www.vim.org/index.php.

[21] I. Viskic and R. Dömer. A flexible, syntax independent representation (SIR) for system level design models. In *Proceedings of EuroMicro Conference on Digital System Design*, August 2006.

CONSTRAINED AND UNCONSTRAINED HARDWARE-SOFTWARE PARTITIONING USING PARTICLE SWARM OPTIMIZATION TECHNIQUE

M. B. Abdelhalim, A. E. Salama and S. E.-D. Habib
Electronics and Communication Department,
Faculty of Engineering, Cairo University, Egypt.

Abstract: In this paper we investigate the application of the Particle Swarm Optimization (PSO) technique for solving the Hardware/Software partitioning problem. The PSO is attractive for the Hardware/Software partitioning problem as it offers reasonable coverage of the design space together with O(n) main loop's execution time, where n is the number of proposed solutions that will evolve to provide the final solution. We carried out several tests on a hypothetical, relatively-large Hardware/Software partitioning problem using the PSO algorithm as well as the Genetic Algorithm (GA), which is another evolutionary technique. We found that PSO outperforms GA in the cost function and the execution time. For the case of unconstrained design problem, we tested several hybrid combinations of PSO and GA algorithms; including PSO then GA, GA then PSO, GA followed by GA, and finally PSO followed by PSO. The PSO algorithm followed by another PSO round gave the best result as it allows another round of domain exploration. The second PSO round assign new randomized velocities to the particles, while keeping best particle positions obtained in the first round. We propose to name this successive PSO algorithm as the Re-excited PSO algorithm. The constrained formulations of the problem are investigated for different tuning or limiting design parameters constraints.

Keywords: Embedded systems, Hardware/Software co-design, Hardware/Software Partitioning, Particle Swarm Optimization, Genetic Algorithm, Evolutionary Algorithms, re-excited PSO.

Abdelhalim, M.B., Salama, A.E., Habib, S.E.-D, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 207–220.

1. INTRODUCTION

Embedded systems typically consist of application specific hardware parts, i.e. FPGAs or ASICs, and programmable parts, i.e., processors like DSPs or ASIPs. In comparison to the hardware parts, the software parts are much easier and faster to develop and modify. Thus, software is less expensive in terms of cost and development time. Hardware, however, provides better performance. For this reason, a system designer's goal is a system which minimizes the weighted sum of the software delay cost, Hardware area cost, and power consumption cost. The weights are determined by the user according to the design's critical parameters.

Hardware/software co-design deals with the problem of designing embedded systems, where automatic partitioning is one key issue. This paper describes a new approach for hardware/software partitioning for single-processor systems. This approach is based on the Particle Swarm Optimization (PSO) algorithm. To further improve the quality of the result obtained by the PSO algorithm, a successive iterative approach is described in this paper that finds a near global optimum solution in a small time. To the best of our knowledge, there is no work based on PSO for HW/SW partitioning problem, apart from our precursor paper [1], limited to the unconstrained case.

The outline of the paper is as follows: Section 2 gives an overview of related work in the field of Hardware/Software Partitioning. In Section 3, a brief summary of the Particle Swarm Optimization algorithm is presented. In Section 4, the implementation of the algorithm and the results are discussed and compared with the results of the Genetic Algorithm, also the concept of re-excited PSO is presented. Section 5 extends the PSO algorithm for constrained HW/SW partitioning problems. The paper conclusions are given in Section 6.

2. HARDWARE/SOFTWARE PARTITIONING

The most important challenge in the embedded system design is partitioning; i.e. deciding which components of the system should be implemented in hardware and which ones in software. Finding an optimal partition is hard because of the large number and different characteristics of the components that have to be considered.

Traditionally, partitioning was carried out manually [2]. However, because of the increase of complexity of the systems, many research efforts have been undertaken to automate the partitioning as much as possible. The suggested partition approaches differ significantly according to the definition

they used to the problem. One of the main differences is whether to include other tasks (such as scheduling where starting times of the components should be determined) [3-4] or just map components to hardware or software only [5-6]. Some formulations assign communication cost to links between hardware and/or software units [7]. The system to be partitioned is generally given in the form of task graph, the graph nodes determined by the model granularity, i.e. the semantic of a node. The node could represent a single instruction, short sequence of instructions [8], basic block [9] or a function or procedure [10-11]. A flexible granularity may also be followed where a node can represent any of the above [5, 12]. Regarding the suggested algorithms, one can differentiate between exact and heuristic methods. The proposed exact algorithms include branch-and-bound [13], dynamic programming [6], and integer linear programming [10]. Due to the slow performance of the exact algorithms, heuristic-based algorithms are proposed. In particular, Genetic algorithms are widely used [7,14] as well as simulated annealing [12, 15], hierarchical clustering [5], and Kernighan-Lin based algorithms such as in [14]. Less popular heuristics are used such as Tabu search [15] and greedy algorithms [16]. Some researchers used custom heuristics, such as MFMC [14], GCLP [17], process complexity [18], and the expert system presented in [3].

3. PARTICLE SWARM OPTIMIZATION

Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Eberhart and Kennedy in 1995 [19-20]. The PSO algorithm is inspired by social behavior of bird flocking or fish schooling. In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles. PSO has been successfully applied in many areas. A good bibliography of PSO applications could be found in [22].

As stated before, PSO simulates the behavior of bird flocking. Suppose the following scenario: a group of birds are randomly searching for food in an area. There is only one piece of food in the area being searched. All the birds do not know where the food is. During every iteration, they learn via their inter-communications, how far the food is. So the best strategy to find the food is to follow the bird which is nearest to the food [21].

PSO learned from this bird-flocking scenario, and used it to solve the optimization problems. Each single solution (particle) is perceived as a “bird” in the search space. Each particle has a fitness value which is evaluated by the fitness function (the cost function to be optimized), and has

a velocity which directs its flight. The particles fly through the problem space by following the current optimum particles.

PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. During every generation (iteration), each particle is updated by following two "best" values. The first one is the position vector of the best solution (fitness) this particle has achieved so far. The fitness value is also stored. This position is called ***pbest***. Another "best" position that is tracked by the particle swarm optimizer is the best position, obtained so far, by any particle in the population. This best position is the current global best and is called ***gbest***.

After finding the two best values, the particle updates its velocity and position according to equations (1) and (2).

$$v_{k+1}^{i} = wv_{k}^{i} + c_1 r_1 (pbest^{i} - x_{k}^{i}) + c_2 r_2 (gbest_{k} - x_{k}^{i}) \quad (1)$$

$$x_{k+1}^{i} = x_{k}^{i} + v_{k+1}^{i} \quad (2)$$

Where v_{k+1}^{i} is the velocity of particle number (i) at the $(k+1)^{th}$ iteration, x_{k}^{i} is the current particle solution (or position). (r_1) and (r_2) are random numbers between 0 and 1. (c_1) is the self confidence (cognitive) factor; (c_2) is the swarm confidence (social) factor. (w) is the inertia factor [21].

The 1st term in equation 1 represents the effect of the inertia of the particle, the 2nd term represents the particle memory influence, and the 3rd term represents the swarm influence.

The velocities of the particles on each dimension may be clamped to a maximum velocity V_{max}, which is a parameter specified by the user. If the sum of accelerations causes the velocity on that dimension to exceed V_{max}, then this velocity is limited to V_{max} [21]. Another type of clamping is to clamp the position of the current solution to a certain range in which the solution has a meaning; otherwise the solution is meaningless [21]. In this paper, position clamping is only applied with no limitation on the velocity values.

In [1] a comparison between PSO and GA showed that PSO is better than GA in that it has small number of tuning parameters, and it is faster than GA due to its main loop's linear complexity.

4. IMPLEMENTATION AND RESULTS

The PSO algorithm is written as a script in the MATLAB program environment [22]. The input to the scripts is a design that consists of the number of nodes. Each node is associated with cost parameters. For experimental purpose, these parameters are randomly generated.

The used cost parameters are:

A Hardware implementation cost (*HWCost)*; which is the cost of implementing that node in hardware (e.g. hardware area and delay, but in most cases the hardware delay is ignored with respect to the software delay).

A Software implementation cost (*SWCost)*; which is the cost of implementing that node in software (e.g. Processing delay and CPU area, the CPU area is ignored as it is independent of the problem size).

This above cost values are uniformly and randomly generated in the range from 1 to 99 [14].

A Power implementation cost (*POWERcost)*; which is the power consumption of the node's implementation. This cost is uniformly and randomly generated in the range from 1 to 9. We use a different range for the Power consumption cost to test the addition of other cost terms with different range characteristics (the cost term will be normalized in the cost function).

The communications cost is included in the hardware and software costs of the nodes, as communications between nodes are already known from the design itself [14].

Consider a design that consists of **m** nodes. A possible solution (particle) is a vector of **m** elements, where each element is associated to a given node. The elements assume a "0" value (if the node is implemented in hardware) or a "1" value (if the node is implemented in software). There are **n** particles (solutions). The particles are initialized randomly.

The velocity of each node is initialized in the range from (-1) to (1), where a negative velocity indicates that the particle is moving towards 0 and a positive velocity indicates that the particle is moving towards 1.

For the algorithm's main loop, equations 1, 2 are evaluated in each loop. If the particle goes out of the permissible region (position from 0 to 1); it is clamped to the nearest limit by the aforementioned clamping technique. The cost function is evaluated for each particle. The used cost function is the normalized sum of the hardware, software, and power cost of each particle according to equation 3.

$$Cost = 100 * \left\{ \frac{HW\,\mathrm{cost}}{allHW\,\mathrm{cost}} + \frac{SW\,\mathrm{cost}}{allSW\,\mathrm{cost}} + \frac{POWER\mathrm{cost}}{allPOWER\mathrm{cost}} \right\} \tag{3}$$

Where ***allHWcost*** (***allSWcost***) is the maximum hardware (software) cost when all nodes are mapped to hardware (software), and ***allPOWERcost*** is the sum of the power cost of the all-hardware solution and the all-software solution.

According to equations 1 and 2; the particle nodes values can take any real value between 0 and 1. But, as a binary problem, the nodes values must be rounded 1 or 0. Therefore we round the position to the nearest integer (i.e. the node is mapped to hardware if the node position is lower than 0.5 and mapped to software if the node value is greater than 0.5)

The algorithm's main loop is terminated if the improvement in the global best solution ***gbest***; remains less than a predefined value (ε) for a predefined number iterations. This predefined number of the iterations and the value of (ε) are user controlled parameters.

The following experiments are performed on a Pentium-4 PC with 3GHz processor speed, 1 GB RAM and WinXP operating system. The experiments are performed using the MATLAB 7 program [22]. The parameters used for PSO experiments are as follows:

No. of particles (Population size) **n** = 60, No. of design size **m** = 512 nodes, ε = 100 * eps, where eps is defined in MATLAB as a very small (numerical resolution) value that equals 2.2204e-016 [22], for each node, the same cost values are used for the two algorithms, for PSO, $c_1 = c_2 = 2$, *w* starts at 1 and decreases linearly until reaching 0 at the end of the run. Those values are suggested in [23-24].

To measure the efficiency of the PSO algorithm, we compared it with the famous evolutionary Genetic Algorithm (GA) [1]. To get the best results for GA, we followed the suggestions in [14] where the algorithm stops after 50 unchanged iterations, but at least 100 iterations must be performed. For the sake of fair comparison between the two algorithms, the stopping criterion is made the same in PSO. Hence, *w* reaches 0 after 100 runs. The GA parameters are set as follows: Selection rate = 0.5 (best 50% of the population are kept unchanged while the others go under the crossover operators [25]). Mutation rate = 0.05 (randomly selected 5% of the population nodes change their values form 0 to 1 and vice-versa at each iteration). The mating is performed using single point crossover.

The results of the GA and PSO algorithms are shown in Figures 1 and 2 respectively.

As shown in the figures, the initialization is the same, but at the end, the best cost of GA is 143.1 while for PSO it is 131.6. This result represents around 8% improvement in the result quality in favor of PSO. Regarding the algorithm processing time, of PSO terminates after 0.609 seconds while GA terminates after 0.984 seconds. This result represents around 38% improvement in performance in favor of PSO.

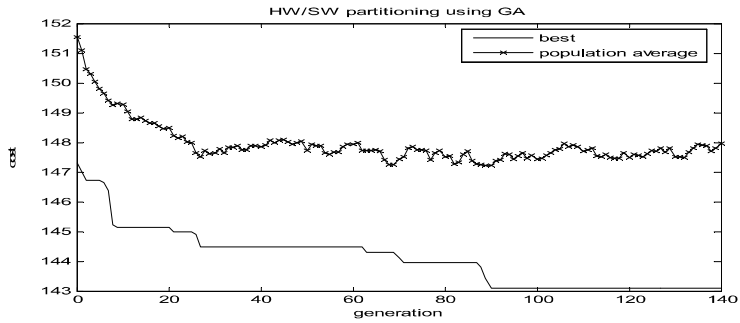


Figure 1. Best cost and Average Population cost of GA

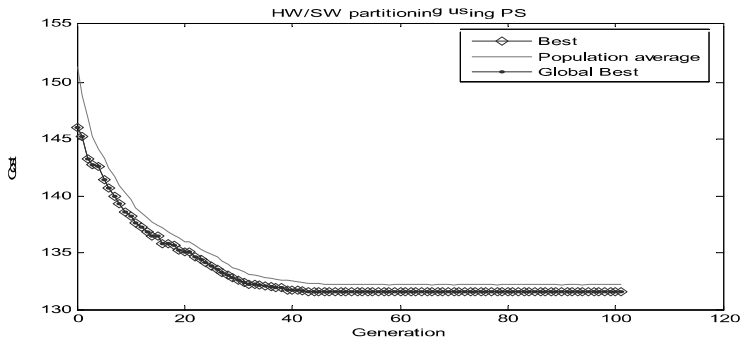


Figure 2. Best cost and Average Population cost of PSO

To further enhance the quality of the results, we tried cascading the two algorithms (GA followed by PSO and PSO followed by GA). This hybrid cascade yielded insignificant improvements [1]. This conclusion motivated us to investigate other methods to improve the quality of the result. The main idea is to cascade the algorithm with itself. Successive GA rounds failed to obtain a better result due to limitations in the algorithm itself [1]. Successive rounds of PSO yielded good results as is discussed in the following subsection.

4.1 Successive PSO (Re-excited PSO) Algorithm

As the PSO proceeds, the effect of the inertia factor (w) is decreased until it reaches 0. Therefore, v_{k+1}^{i} at the end depends only on the particle memory influence and the swarm influence (2nd and 3rd terms in equation 1). Hence, the algorithm may converge to local optimum positions. We thus propose to take the run's final results (particles positions) and start allover again with (w) = 1 and re-initialize the velocity (v) with new random values. We found that the result quality is improved with each new round until it settles around a certain value. Figure 3 plots the best cost in each round. The curve starts

with cost ~133 and settles at round number 30 with cost value ~116.5 which is significantly below the results obtained in the previous subsection. The program performed 100 rounds but it could be modified to stop much earlier if the result remains unchanged for a certain number of rounds.

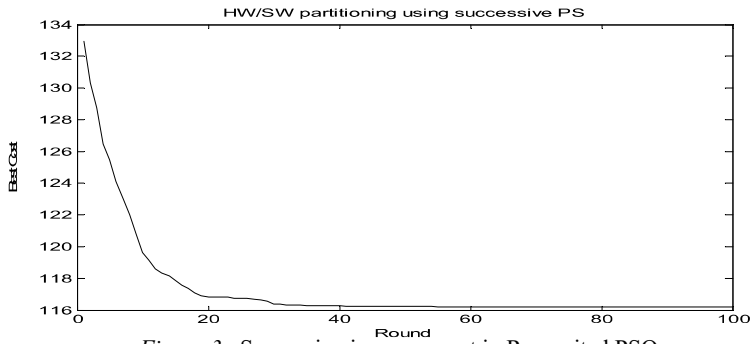


Figure 3. Successive improvement in Re-excited PSO

As the new algorithm depends on re-exciting new randomized particle velocities at the beginning of each round, while keeping the particle positions obtained so far, it allows another round of domain exploration. We propose to name this successive PSO algorithm as the *Re-excited PSO algorithm.* In nature, this algorithm looks like giving the birds a big push after they have settled in their best position. This push re-initializes the inertia and speed of the birds so they are able to explore new areas, unexplored before. Hence, if the birds find a better place, they will go there, otherwise they will return back to the place from which they were pushed. This re-excited PSO algorithm can be viewed as a variant of the re-start strategies for PSO published elsewhere [26-29]. However, re-excited PSO algorithm is not identical to any of these previously published re-start PSO algorithms.

4.2 Comparison for Different Design Sizes

To make sure that our results are not affected by the design size, we considered different designs with size ranging from 5 nodes to 1020 nodes. We used the same parameters as described in Section 4 and we ran the algorithms on each design size 10 times and took the average results. The stopping criterion of the re-excited PSO is to stop when the best result remains the same for 10 successive rounds. Figure 4 represents the design quality improvement of PSO over GA, re-excited PSO over GA, and re-excited PSO over PSO. It shows that, on the average, PSO outperforms GA by an average of 7.8%. On the other hand, re-excited PSO outperforms GA

by an average of 17.4%, and o utperforms normal PSO by an average of 10.5%.

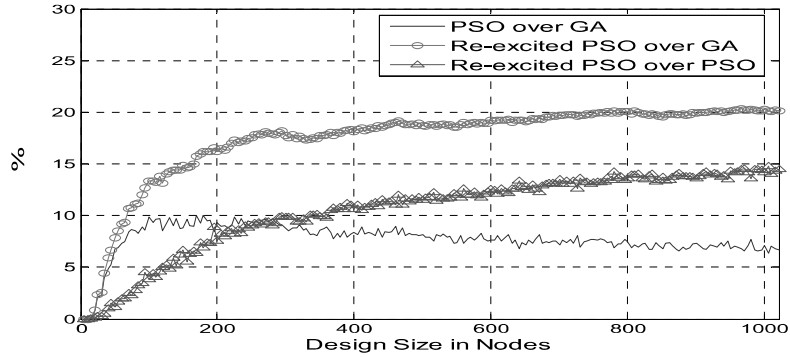


Figure 4. Quality improvement percentage for different design sizes

Figure 5 represents the speed improvement of PSO over GA for different design sizes (original and fitted curve, the curve fitting is done using MATLAB's Basic Fitting tool). Figure 5 shows that, on the average, PSO outperforms GA by a ratio 29.3% improvement in speed.

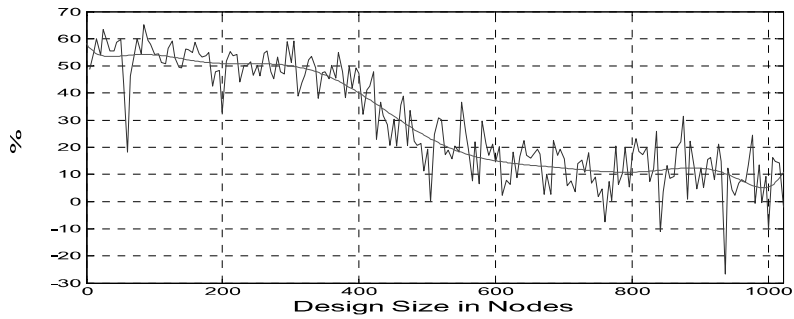


Figure 5. Speed improvement percentage for different design sizes

From Figure 5, the speed improvement is high for small designs (from 40% to 60%), constant for large designs (around 10%), and decreases linearly from 40% to 10% for medium-size designs.

5. CONSTRAINED PROBLEM FORMULATION

In embedded systems, the constraints play an important role in the success of a design, where hard constraints mean higher design effort and therefore a high need for automated tools to guide the designer in critical design decisions. In most cases, the constraints are mainly the software deadline times (for real-time systems) and the maximum allowable area for

hardware. For simplicity, we will refer to them as software constraint and hardware constraint respectively.

Mann [14] divided the hardware software partitioning problem into 5 sub-problems (P1 – P5). The unconstrained problem (P5) is discussed in Section 4. P1 deals with both Hardware and Software constraints. P2 deals with hardware constrained designs while the software cost is unconstrained. P3, in the other hand, deals with software constrained designs while the hardware cost is unconstrained. Finally, P4 minimizes HW/SW communications while satisfying hardware and software constraints.

The constraints affect directly the cost function. Hence, equation 3 should be modified to account for constraints violations. Lopez-Vallejo[3] suggested three different techniques for the correction function formulation:

Mean Square Error minimization: This technique is useful for forcing the solution to meet certain equality, rather than inequality, constraints.

The general expression for Mean Square Error based cost function is:

$$\mathrm{MSE\_cost_i} = \mathrm{k_i} * \frac{(\mathrm{cost_i} - \mathrm{constraint_i})^2}{\mathrm{constraint_i}^2} \tag{4}$$

Where $\mathrm{constraint_i}$ is the constraint on parameter i, $\mathrm{k_i}$ is a weighting factor, and $\mathrm{cost_i}$ is the parameter cost value.

Penalty Methods: These methods punish the solutions that produce medium or large constraint violations, but allow solutions close to the boundaries defined by the constraints [3]. The penalty methods may produce invalid solutions with better overall cost than valid ones. The cost function in this case is formulated as:

$$Cost(x) = \sum_i k_i * \frac{\mathrm{cost}_i(x)}{Total\mathrm{cost}_i} + \sum_{ci} k_{ci} * viol(ci, x) \tag{5}$$

Where $\mathrm{k_i}$ and $\mathrm{k_{ci}}$ are weighting factors (100 in our case), i denotes the design parameters, ci denotes the constrains-violated parameters, viol(i) is the correction function of the constrains-violated parameters. viol(i) could be MSE_cost (equation 4) [3] or its square root [14], or any other suitable form. Note that viol(i) will be zero if the constrained parameter is not violated.

Barrier Techniques [3]: which forbid the exploration of solutions outside the allowed design space by adding a high cost term to all invalid solutions. There are two forms of the barrier techniques. The first form assigns a constant high cost to all invalid solutions (for example infinity). The second form adds an extra constant base cost term to all invalid solutions. This second form is obviously superior to the first form[14]. The base cost term of the second form can be a constant larger than any cost

value produced by any valid solution. In our case, and since our cost function is normalized, we select the constant base cost to be "one" for each violation term (“one” for hardware violation, “one” for software violation … etc.)

In order to determine the best method to be adopted, a comparison between the penalty methods (MSE_cost or its square root correction function) and the barrier methods (infinity cost vs. constant base cost) is performed.

For double constraints problem (P1), generating valid initial solutions is hard and time consuming, and hence, the barrier methods should be ruled out for such problems.

When dealing with single constraint problems (P2 and P3), one can use the Fast Greedy Algorithm (FGA) [14] to easily generate valid initial solutions. FGA starts by assigning all nodes to the unconstrained side. It then proceeds by randomly moving nodes to the constrained side until the constraint is violated.

Our experiments showed that combining the constant base cost barrier method with any penalty method (MSE_cost square term gives slight improvements over its square root) gives higher quality solutions and guarantees that no invalid solutions beat valid ones as the main disadvantage of the Penalty methods is that they admit invalid solutions.

In all experiments, all HW (SW) constraints are given relative to the area (delay) of an all-hardware (software). In the following experiments, the used cost function takes the combined form shown in equation 6.

$$Cost(x) = \sum_i k_i * \frac{\cos t_i(x)}{Total\cos t_i} + \sum_{ci} k_{ci}(Penalty\_viol(ci,x) + Barrier\_viol(ci)) \quad (6)$$

5.1 Single constraint experiments

As P2 and P3 are treated the same in our formulation, we consider the software constrained problem (P3) only. Two experiments were performed, the first one with relaxed constraint where the deadline (Maximum) delay is 40% of the delay of an all-software solution. The second one is a hard real-time system where the deadline is 15%. The parameters used are the same as in Section 4. FGA is used to generate the initial solutions and re-excited PSO is performed for 10 rounds. In the cases of GA and normal PSO only, all results are based on averaging the results of 100 runs.

For the first experiment; the average quality of the GA solution is ~ 137.6 while for the PSO it is ~ 131.3 and for the re-excited PSO it is ~ 120. All final solutions are valid since the FGA is used as the initialization scheme.

For the second experiment, the average quality of the GA solution is ~ 147 while for the PSO it is ~ 137 and for the re-excited PSO it is ~ 129.

5.2 Double constraints experiments

When testing P1 problems, the same parameters as Section 4 are used except that FGA is not used. Two experiments were performed: balanced constraints where hardware area and software delay constraints are 45%. The other one is an unbalanced-constraints problem where hardware constraint is 60% and the software delay constraint is 20%.

For the first experiment; the average quality of the solution of GA is ~ 158 and invalid solutions are obtained during 22 of the runs. The best valid solution cost was 137. For PSO, the average quality is ~ 131 with valid solutions during all the runs. The best valid solution cost was 128.6. Finally for the re-excited PSO; the final solution quality is 119.5.

For the second experiment, the average quality of the GA solution is ~ 287 and no valid solution is obtained during the runs (100 is added to the cost as a constant base penalty) as there is always violations in delay constraints. For PSO the average quality is ~ 251 with no valid solution is obtained during the runs as there is always violations in delay constraints. Finally for the re-excited PSO, the final solution quality is 125 (a valid solution is found in the seventh round). This indicates the power of re-excited PSO over both PSO and GA for hard constrained problems.

6. CONCLUSIONS

In this paper we investigated the application of Particle Swarm Optimization (PSO) technique for the Hardware/Software partitioning problem. We tested the PSO algorithm against the Genetic Algorithm (GA). Averaged over different design sizes ranging from 5 nodes to 1020 nodes; The PSO outperforms GA by a ratio of 7.8% improvements in the result quality and 29.3% speed improvement. The hybrid algorithms where one algorithm is cascaded with the other were tested. Hybrid cascading of GA and PSO proved fruitless. The best performance is obtained when one PSO round is cascaded by another PSO round with new randomized particle velocities, while keeping best particle positions obtained in the first round. We propose to name this successive PSO algorithm as the re-excited PSO algorithm. The quality of re-excited PSO solutions outperforms both GA and normal PSO design qualities by a ratio of 17.4% and 10.5% respectively.

The constrained problem is also investigated in the cases of tuning constraints and limiting constraints. The cost function is modified in each

case to accommodate the problem under investigation. Our experiments showed that re-excited PSO outperforms both PSO and GA. For hard unbalanced constrained problems, the re-excited PSO succeeded in finding a valid solution, while PSO and GA alone failed to obtain a valid solution.

The PSO features gradual exploration of the design space as the particles fly through the design space guided by the weighted sum of their inertia, memory and the flock communications. No jumps take place. This gradual exploration of the design space enables the PSO to detect steep local minima that are hard to detect by other algorithms. Also, PSO is blessed with a linear complexity of its main loop. Hence, PSO is attractive for large Hardware/ software partitioning problems.

REFERENCES

[1] M. B. Abdelhalim, A. E. Salama and S. E.-D. Habib, "Hardware-Software Partitioning Using Particle Swarm Optimization Technique", Proceedings of 6th Int. Workshop on System-On-Chip for Real-Time Applications, Cairo, Egypt, pp. 189 – 194, 2006.

[2] P. L. Marrec, C. A. Valderrama, F. Hessel, A. A. Jerraya, M. Attia, and O. Cayrol, "Hardware, software and mechanical cosimulation for automotive applications", Proceedings of 9th Int. Workshop on Rapid System Prototyping, Leuven, Belgium, pp. 202 – 206, 1998.

[3] M. Lopez-Vallejo and J. C. Lopez, "On the hardware-software partitioning problem: system modeling and partitioning techniques", ACM transactions on design automation for electronic systems, Volume 8, Issue 3, pp. 269 – 297, July 2003.

[4] B. Mei, P. Schaumont, and S. Vernalde, "A hardware/software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems", Proceedings of 11th ProRISC, Veldhoven, Netherlands, 2000.

[5] F. Vahid, "Partitioning Sequential Programs for CAD using a Three-Step Approach", ACM Transactions on Design Automation of Electronic Systems, Volume 7, Issue 3, pp. 413 – 429, July 2002.

[6] J. Madsen, J Gorde, P. V. Knudsen, M. E. Petersen, and A. Haxthausen, "LYCOS: The Lyngby co-synthesis system", Design Automation of embedded Systems, Volume 2, Issue 2, pp. 195 – 236, April 1997.

[7] N. K. Jha and R. P. Dick, "MOGAC: a multiobjective genetic algorithm for hardware-software co-synthesis of distributed embedded systems", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume 17, Issue 10, pp. 920 – 935, Oct. 1998.

[8] G. Stitt, F. Vahid, G. McGregor, and B. Einloth, "Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decoder", IEEE/ACM CODES+ISSS'05, New York, USA, pp. 285 – 290, 2005.

[9] P. V. Knudsen, and J. Madsen, "PACE: a dynamic programming algorithm for hardware/software partitioning", 4th Int. Symposium on Hardware/Software Co-Design, Pittsburgh, Pennsylvania, USA, pp. 85 – 92, 1996.

[10] M. Ditzel. "Power-aware architecting for data-dominated applications", PhD thesis, Delft University of Technology, 2004.

[11] J.R. Armstrong, P. Adhipathi, and J.M. Baker, Jr. "Model and synthesis directed task assignment for systems on a chip", 15th Int. Conf. on Parallel and Distributed Computing Systems, Cambridge, USA, 2002.

[12] J. Henkel, R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques", IEEE Transactions on Very Large Scale Integration Systems, Volume 9, Issue 2, pp. 273 – 289, 2001.

[13] N. N. Binh; M. Imai, A. Shiomi, and N. Hikichi, "A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate counts", Proceedings of 33rd Design Automation Conf., Las Vegas, Nevada, USA, pp. 527 – 532, 1996.

[14] Z. A. Mann, "Partitioning algorithms for Hardware/Software Co-design", PhD thesis, Budapest University of Technology and Economics, Hungary, 2004.

[15] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/software partitioning besed on simulated annealing and tabu search", Design automation for embedded systems, Volume 2, Issue 1, pp. 5 – 32, Jan. 1997.

[16] K. S. Chatha, and R. Vemuri, "MAGELLAN: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs", Proceedings of the 9th Int. Symposium on Hardware/Software Codesign, Copenhagen, Denmark, pp. 42 – 47, 2001.

[17] A. Kalavade and E. A. Lee. "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem", 2nd Int. Symposium on Hardware/Software Codesign, Grenoble, France, pp. 42 – 48, 1994.

[18] P. Adhipathi, "Model based approach to Hardware/Software Partitioning of SOC Designs", MSc Thesis, Virginia Polytechnic Institute and State University, 2004.

[19] J. Kennedy, and R.C. Eberhart, "Particle swarm optimization", Proceedings of IEEE int. Conf. on Neural Networks, Volume. 4, Perth, Australia, pp. 1942 – 1948, 1995.

[20] R.C. Eberhart, and J. Kennedy, "A new optimizer using particle swarm theory", Proceedings of the 6th int. symposium on micro-machine and human science, Nagoya, Japan, pp. 39-43. 1995.

[21] www.swarmintelligence.org

[22] www.mathworks.com

[23] Y. Shi, and R. Eberhart, "Parameter selection in particle swarm optimization", Proceedings of 7th Annual Conf. on Evolutionary Computation, New York, USA, pp..591 – 601, 1998.

[24] Y. L. Zheng, L. H. Ma, L. Y. Zhang, and J. X. Qian, "On the convergence analysis and parameter selection in particle swarm optimization", Proceedings of the 2nd Int. Conf. on Machine Learning and Cybernetics, Xi-an, China, Volume 3, pp. 1802 – 1807, 2003.

[25] R. L. Haupt, and S. E. Haupt, "Practical Genetic Algorithms", Second Edition, Wiley Interscience, 2004.

[26] M. Settles and T. Soule. "A hybrid GA/PSO to evolve artificial recurrent neural networks" Intelligent Engineering Systems through Artificial Neural Networks, volume 13, pp. 51 – 56, St. Louis, USA. 2003. ASME Press.

[27] J. Tillett, T.M. Rao, F. Sahin, and R. Rao, "Darwinian particle swarm optimization", Proceedings of the 2nd Indian Int. Conf. on Artificial Intelligence, Pune, India , pp. 1474– 1487, 2005.

[28] S. Pasupuleti and R. Battiti, "The Gregarious Particle Swarm Optimizer (GPSO)", Genetic And Evolutionary Computation Conf., Seattle, USA, pp. 67 – 74, 2006.

[29] F. Van den Bergh, " An Analysis of Particle Swarm Optimizer.", PhD thesis, Department of Computer Science, University of Pretoria, Pretoria, South Africa, 2002.

USING ASPECT-ORIENTED CONCEPTS IN THE REQUIREMENTS ANALYSIS OF DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

Edison P. Freitas[1], Marco A. Wehrmeister[1,3], Carlos E. Pereira[1,2], Flavio R. Wagner[1], Elias T. Silva Jr[1], Fabiano C. Carvalho[1]

[1]*Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brazil*
{epfreitas, mawehrmeister, flavio, etsilvajr, fccarvalho}@inf.ufrgs.br

[2]*Departamento de Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Brazil*
{cpereira}@eletro.ufrgs.br

[3]*Heinz Nixdorf Institute, University of Paderborn, Germany*

Abstract: Distributed Real-time Embedded (DRE) systems commonly have several requirements that are difficult to handle when a pure object-oriented method is used for their development. These requirements are called non-functional requirements and refer to orthogonal properties, conditions, and restrictions that are spread out over the system. In general, the specification of those requirements using pure object oriented methods leads to intermixed specification with the functional requirements. This work presents a proposal to use the concepts of aspect orientation in the specification of DRE requirements at the system analysis phase, offering a link from those requirements to system elements in the design phase. To support our proposal, it was performed an adaptation of a method called FRIDA (From RequIrements to Design using Aspects) to the DRE generating the RT-FRIDA (Real-Time FRIDA).

Key words: Requirements Specification, Distributed Real-time Embedded Systems, Aspect-Orientation, Separation of Concerns.

1. INTRODUCTION

The increasing complexity of distributed real-time embedded (DRE) systems requires new techniques to improve the design in order to allow the

Freitas, E.P., Wehrmeister, M.A., Pereira, C.E., Wagner, F.R., Silva Jr, E.T., Carvalho, F.C, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 221–230.

system evolution, maintainability, and reuse of previously developed artifacts. The complexity in developing DRE systems appears since the early phases, when the requirements are being specified. Much of the problem in this specification is related to the handling of the non-functional requirements (NFR) and their traceability in the following design phases.

What really worries system developers when dealing with NFR is the crosscutting concerns which characterize the main concept related to the problem of handling NFR. If not properly handled, those concerns are responsible for tangling code and leak of cohesion. In the literature, there are several works addressing this subject, such as [1][6]. In order to handle the separation of concerns, several works propose guidelines to separate NFR from the functional ones. Among those works stand out subject-oriented programming [11] and aspect-oriented programming [8]. Both approaches address the problem at the implementation level, however the handling of NFR must be taken into account as soon as possible to enhance the system design. This fact motivates pushing the separation of concerns to the early phases of the design, as in the Early-Aspects [12] approach.

Real-time systems have very important NFR, which is the concern about the predictability in the execution. The complexity related to NFR analysis increases when those systems become distributed and embedded. To deal with these NFR, many proposals suggest the use of aspects [13][15].

This paper presents an approach to deal with the complexity exhibited by NFR in DRE systems by adapting the FRIDA [2] method to the DRE domain, allowing a clear specification of the system requirements, which can be easily mapped into design elements. This paper focuses the requirement specification in the analysis phase of DRE systems.

The remainder of this paper is organized as follows. Section 2 presents the original FRIDA and RT-FRIDA. Section 3 show the use of RT-FRIDA method through a case study. The related work is discussed in section 4. Finally, the final remarks and future work are presented in Section 5.

2. THE FRIDA MODEL

FRIDA is a method that offers a sequence of phases to support requirement analysis and system design. The main goal is to deal with the complexity of NFR separately from the functional ones. The method is based on concepts of the Aspect-Oriented (AO) paradigm [8].

Considering NFR in system analysis is a way of avoiding code tangling and the undesired mixing of different concerns in later design phases, which is always present in systems developed with object-oriented (OO) methods.

The problem with the OO paradigm is that there is no specific element dedicated to handle NFR because only the functional dimension of the system is considered. Using an AO approach, FRIDA tries to fulfill this gap by providing a way to consider both functional and NFR during the specification of systems.

FRIDA is divided in six main phases. Fig. 1 presents the whole method. The first phase is dedicated to identifying the system functional requirements. Use case diagrams and templates are used to elicit those requirements. In the second phase, the NFRs are identified and specified. To perform this task, check-lists, lexicons, and conflict resolution rules are used. A link among classes, actors, and the use cases is created in the third phase. The next phase performs almost the same, but for the NFR, representing them visually in the class diagram. In the fifth phase, the functional requirements are represented by classes that are linked to aspects. Finally, in the last phase, the source code of classes and aspect (i.e. skeletons for classes and aspects) is generated.

2.1 RT-FRIDA: Aplying FRIDA to the DRE Domain

The FRIDA method provides a consistent way to separate non-functional from functional requirements from the early phases of system development, representing a relevant contribution to the system analysis and to the mapping of requirements into design elements [2].

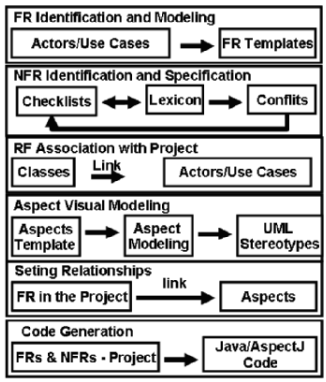


Figure 1. Original FRIDA Method [2]

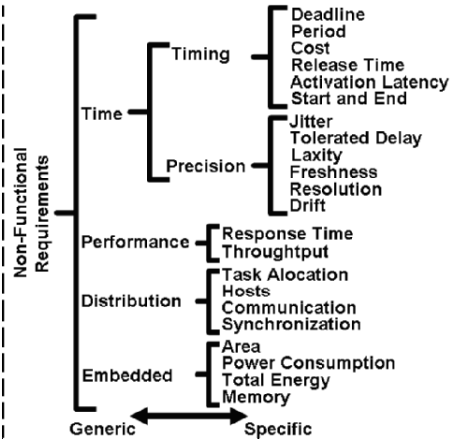


Figure 2. NFRs classification for DRE Systems

FRIDA focuses on the fault tolerance, with a vocabulary and tools designed to support the analysis of fault tolerant systems. In order to adapt FRIDA to the DRE domain, the first step was to identify the concerns related to DRE development. Some requirements of this domain are shown in Fig. 2. Those requirements are based mainly on the study present in [3] and in the

IEEE [7] and SEI [5] glossaries. Based on this classification, some FRIDA tools were adapted. It is important to highlight that many DRE systems also have fault tolerance requirements. Thus every requirement considered in the original method can also be used in the RT-FRIDA.

In order to explain the use of the RT-FRIDA, a case study showing the specification of the requirements of a movement control system of an Unmanned Air Vehicle (UAV) is considered. The following section describes the case study and details each phase of the adapted method.

3. CASE STUDY: USING RT-FRIDA

The case study consists on the design of distributed, real-time and embedded movement control system to equip an Unmanned Air Vehicle (UAV). The UAV consists in a helicopter that can be used in a variety of applications, like environment monitoring, support the rescue of victims of natural disasters, urban security, among many other useful applications. The whole design includes functions like navigation, movement control, collision detection, target pursuit, system maintenance, mission management, camera control and data exchange with an on ground base. More details can be consulted in [4]. The presented case study focus on the movement control system, which is composed by sensors and actuators, and also have interface with the maintenance, navigation and data transfer systems. Its operation consists of: (i) the receiving of new movement parameters from navigation system; (ii) the reading of environment and movement sensors; and (iii) the calculus of the guiding and the piloting parameters that will determine the new values that must be applied over rotor actuators. To achieve those operations with success, the system must meet several real-time requirements related to the rate of sampling data, period and deadline of several tasks, among others. The system has several sensors and actuators spatially separated, thus there is also many distribution requirements that must be observed, like communication between processors and concurrency control over shared data. The main concern about the embedded dimension in this system is the energy monitoring and control. As the system can operate in different modes, that depend on the circumstances (i.e. UAV in danger, the “go-back-home” policy establishes the maximum priority to the movement control in spite of the tasks like camera operation), a energy level monitor and control must adequate the system to meet essential requirements, like in hard real-time tasks.

3.1 Requirements Identification and Specification

The analysis starts with the identification of the functional requirements to build the use case diagram. The first step is to fulfill the templates that specify each identified use case.

At this point, the analysis handles the NFR using a set of check-lists in order to elicit the NFR present in the system. Four check-lists for the DRE domain have been created, covering the following areas: time, performance, distribution, and embedded. Each check-list can have sub-check-lists describing how specific and how generic is the requirement. For instance, a question that appears in the check-list of "embedded" concerns regards the power consumption constraints like system autonomy. An example of a full check-list for "embedded" concern is presented in Fig. 3(a). The first column lists the non-functional requirements and the inferring questions, while the second one means relevancy of the requirement, the third column gives its priority and fourth column gives information about restrictions, conditions, and/or a description of the requirement. Additionally, other check-lists for each generic NFR presented in Fig. 2 were created. Due to space restrictions, only one check-list is presented.

	Rel	Pr	R/C/D
Embedded			
Area			
Is there any restriction in relation to the area used by a system component?			
Power consumption			
Is there any restriction about the power consumption of any system component?	X	10	Consumption optimization according to the operation mode.
Is there a need of controlling the use of the energy in the system?	X	10	Even in a non-special operation mode, the consumption must be monitored and controlled.
Total Energy			
Is any restriction about the available energy?			
Is there an estimate of how long the energy must last?			
Is there any alternative energy source?	X	3	Solar energy can be used.
Memory			
Is there any restriction about data storage?			
Is there restriction about the use of memory to the running code?			

Figure. 3a. Check-list example

<NFR_generic> ::= <time> | <performance> | <distribution> | <embedded>

<embedded> ::= <area> | <power_consumption> | <total_energy> | <memory>

<area> ::= <n> logic cells | <n> <lenght_unit>^{2}

<power_consumption> ::= <n> Watts | <n> Joules by operation

<total_energy> ::= <n> Joules | <n> Joules by component | <autonomy>

<autonomy> ::= <n> of operations | <n> de operations by <time_unit> | <n> <length_unit> covered | <n> <time_unit>

<memory>::= <n> <data_unit> of storage | <n> <data_unit> permanent memory | <n> <data_unit> of non-permanent memory | requires <n> <data_unit>

<length_unit> ::= km | m | dm | cm | mm | kilometer | meter| decimeter | centimeter| millimeter

<n> ::= <n> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9|,

<time_unit> ::= h | min | s | ms | µs | ns | hour | minute ||second | millisecond | microsecond | nanosecond | day | week | month | year

<data_unit> ::= bit | byte | Kbit | Kbyte | Mbyte | Tbyte | Megabit | Megabyte | Gigabit | Gigabyte | Terabyte

Figure. 3b. Lexicon example

After the use of check-lists, it may happen that a given NFR could not be satisfactorily specified or even could not be identified at all. To refine the identification and specification of NFR, a lexicon is used. This lexicon

consists of rules organized in *Backus Naur Form* (BNF) [9] that can be used to parse descriptions (in natural language) of the system. An example for embedded requirements can be seen in Fig. 3(b). In the same way that was done for check-lists, specific lexicons for each generic NFR were created.

The next step is to identify conflicts among NFR through the creation of a matrix with all identified NFR. If a requirement conflicts with another one, the cell in the matrix that meets both requirements is checked signalizing the conflict. The priorities defined in the check-lists are used to solve the identified conflicts. If two or more conflicting requirements have the same priority, the stakeholders must be consulted in order to solve them.

After the conflicts removal and the decision about which NFR will be considered in the system development, the next step is to fulfill a template for each NFR. The use of one NFR template is shown in Figure 4. The column "Item" describes the evaluated NFR feature. The column "Description" gives the meaning of each entry and the "Case Study" column gives an NFR example from the presented UAV case study. This example shows the specification of the "energy" feature of the "embedded" NFR.

	Item	Description	Case Study
Identification	Identifier	An identification that will allow the traceability of the concern over the whole project.	NFR-8
	Name	Crosscutting concern's name.	Power Consumption Monitoring
	Author	The responsible for the concern identification and definition.	Thomas Alva Edison
Specification	Classification	Class to which the concern belongs.	Embedded/Power Consumption
	Description	Description of how the concern affects system functionalities.	The system consumes energy to perform its activities. This consume must be measured by each activity and controlled to achieve an optimized consumption.
	Affected Use Cases	List of the use cases affected by the concern.	(1) Environment Sensing; (2) Rotor Sensing; (3) Piloting; (4) Guidance. (5) Helicopter Movement Control; (6) Signal Alarm
	Context	Determines when the concern is expected to affect a use case.	Each time an activity is performed, the current level of energy must be measured before and after the activity execution.
	Scope	(Global/Partial) The requirement is global if it affects the whole system, and it is partial if affects only a part of the system.	Global
Decision and Evolution	Priority	A number used to decide the relative importance among non-functional concerns.	10
	Status	Requirement possible status are: 0 - identified; 1 - analysed; 2 - specified; 3 - approved; 4 - cancelled; 5 - finished;	5

Figure 4. The template used to specify NFR

The final step in this phase is to complete the use case diagram with the considered NFR. As stated before, this paper focuses on the movement control system, however all the expected functionality of the UAV movement control is shown in Figure 5. As can be observed, some functions have NFR affecting their behavior. Those NFR affect different functions, which will certainly imply a decentralized handling in the final system (making harder the reuse and maintainability). In this case study, we

consider concerns about timing, precision, distribution and embedded. The first one has two facets: the timing control that handles the execution of the activities, and the timing parameters that handle all information about time constraints. The second one deals with how imperative is to meet time requirements. The third handles the distribution problem, in this case specifically with the synchronization that must exist in the concurrent accesses to data stored in different nodes, and task allocation over different nodes. The forth is the embedded concern that is related to the energy consumption of the system.

The notation used in this work is not standardized by the OMG. It follows ideas taken mainly from [1]. In Figure 5, it can be seen how the NFR explained above affect the desired system functionality, where each affected functionality (represented through use cases) is annotated with a stereotype that represents the corresponding NFR that affects it. As can be seen in the Figure 5, a number of use cases are affected by several NFR. It is important to highlight that several NFR can affect more than one functionality.

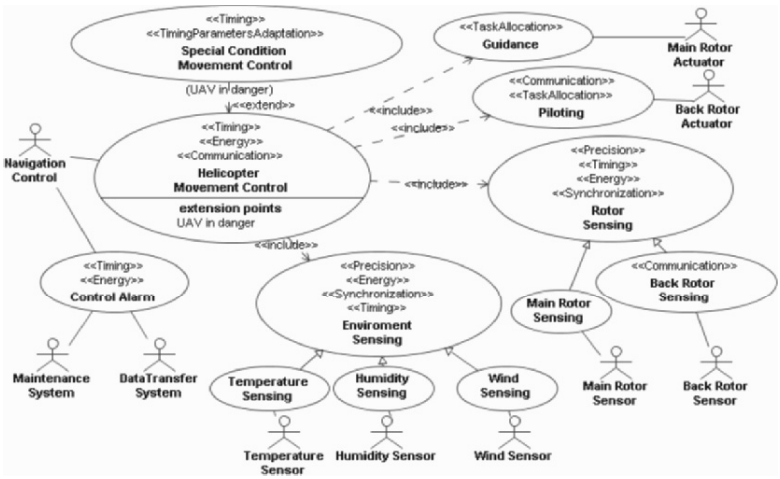


Figure 5. The use case diagram of the UAV movement control with NFR.

3.2 Requirement Association with Project Elements

In this phase, the designer maps the requirements (identified and specified in the first phase) with elements that take part in the system design. There are three main tasks that have to be performed in this phase:

- Extract from the use case diagram and from the templates of functional requirements the concepts and attributes that will compose the system functional part. This consists of classes that will be detailed in the design phase;

- Extract the aspects from the information contained in the use case diagram and the NFR templates. This information will define the aspects that will handle each NFR. It is important to notice that in this phase the developer has an aspect framework called DERAF [16] that contains a set of aspects capable to handle DRE NFR;
- Composition of previously extracted information into a mapping table that will link the requirements with the project elements. This table is very important to guarantee the traceability of requirements over the system life cycle. Additionally, this table relates the functional requirements with non-functional ones that affect them.

		Non-Functional Requirements				Classes responsible for handling FRs
		Periodicity	Energy Monitoring	...	NFR n	
Functional Requirements	Helicopter Movement Control	X	X			ControlSubSystem
						MovementControl
	Movement Sensing	X	X	...		MovementSensing SubSystem
						MovementEncoder
						MovementInformation
	...				...	...
	FR n					Class n
Aspects responsible for handling NFRs		Periodic Timing	Energy Monitoring Energy Control	...	Aspect n	

Figure 6. The mapping table relating FRs to classes and NFRs to aspects

The mapping table shown in Figure 6 is organized as follows: NFR are set in the top row; function requirements are set in the first column from the left side. Aspects that handle a specific NFR are set in the bottom row in the corresponding column, while classes that handle functional requirements are set in the column in the right side of the table, in the corresponding row. Cells relating functional requirements that are affected by NFR are marked with an "X". It is important to highlight that as well as some functional requirements can be handled by more than one class, NFR can also be handled by more than one aspect.

3.3 Design Phase

The design phase is not the focus of this paper, but as it is a part of the RT-FRIDA, this section will draw an overview of some issues in this phase. Design phase consists of the construction of diagrams that represent the proposed solution to the system. With the information retrieved from the mapping table and from the templates, the designer can construct the class diagram that represents the structural of the elements that are responsible to handle the functional requirements. The method uses also RT-UML (i.e. UML diagrams annotated with the real-time profile [10]) notation to improve the richness of the used diagram.

In order to represent the non-functional dimension, the method proposes the use of two different diagrams: (i) the Aspect Crosscutting Overview Diagram (ACOD), which shows the aspects affecting classes; and (ii) the Join Point Designation Diagram (JPDD) [14], which emphasizes points in which functional elements (e.g. class, message exchange, etc) are affected by an aspects. More details about the design phase can be seen in [16].

4. RELATED WORK

The aspect-orientated paradigm is a relatively new concept, however there are some proposals to use it in DRE systems, especially to handle real-time requirements. The majority of the works in this domain propose the use of aspects in the implementation phase, like the approach presented in [15].

Another remarkable work is [13]. This work proposes a set of tools named VEST (Virginia Embedded System Toolkit) that uses aspects to compose a new DRE system based on a component library. Even being an important work, it does not present a concern about the separation of requirements in the analysis level, but only in the design and implementation.

Some proposals bring the concept of aspects to early phases of a system development, like [1] and [17]. Those proposals had influenced the present work. The first one proposes the use of aspects in requirements analysis and its notation in UML use cases, but does not provide a link with design elements. The second proposal describes a way to separate functional and non-functional requirements in the system structure, but besides of it does not consider the analysis phase, it proposes the use of UML diagrams with much text-based information in tags to explain how to handle NFR.

5. FINAL REMARKS AND FUTURE WORK

This paper proposes the use of aspect-orientation to develop high quality DRE systems using an adapted version of the FRIDA method. By adapting a well-defined aspect-oriented method to the DRE domain, like FRIDA, the goal is to provide efficient tools to analyze and model NFR of this domain. This separation of concerns from early phases of development allows a better understanding of system complexity and also a better base to build its structure. It can improve the reuse of system components, because each NFR can be handled by specific components in the system design, avoiding the intermixed specification of system elements.

As future work, the authors intend to incorporate the RT-FRIDA method into a CASE tool to assist the system development, as well as to assist the

designer to obtain metrics about the quality of the design at earlier stages. Another ongoing activity is the definition and implementation of a framework of aspects (DERAF) to handle NFR during modeling and implementation phases of DRE systems.

REFERENCES

1. Araújo, J., Moreira, A., Brito, I., Rashid, A. "Aspect-Oriented Requirements with UML", Workshop on Aspect-oriented Modeling with UML, UML (2002), Dresden, Germany.
2. Bertagnolli, S. C., Lisbôa, M. L. B. "The FRIDA Model", In: Analysis Aspect-Oriented Software, Germany, (Held in conjunction with ECOOP 2003).
3. Burns, A., Wellings, A. Real-time systems and programming languages, Addison-Wesley, 2nd edition (1997).
4. Cai, G., Peng, K. , Chen, B. M., Lee, T. H. "Design and Assembling of a UAV Helicopter System", Proc. of the International Conference on Control and Automation, (ICCA2005), IEEE Computer Society, (2005), pp. 697-702.
5. Carnegie Mellon Software Engineering Institute, "Online Technical Terms Glossary", http://www.sei.cmu.edu/str/ indexes/glossary/, Sep. 2006
6. Chung, L. and Nixon, B.A. "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach", In: Proc. of 17th International Conference on Software Engineering, ACM Press, pp. 25-37 (1995).
7. Institute of Electrical and Electronics Engineering (2006), "IEEE Standard Glossary", http://standards.ieee.org/catalog/olis/arch_se.html
8. Kiczales, G., et al., "Aspect-Oriented Programming", In: Proc. of ECOOP, Lecture Notes in Computer Science 1241, Springer-Verlag, pp. 220-240 (1997).
9. Naur, P., Backus, J. W. "Revised Report n the algorithmic Language Algol 60", (1969) Programming Systems and Languages, Edited by Saul Rosen, New York, McGraw-Hill .
10. Object Management Group, "UML profile for Schedulability, Performance and Time", v.1.1, www. omg.org/cgi-bin/doc?formal/2005-01-02, Sep. 2006
11. Ossler, H., Tarr, P. (1999) "Using subject-oriented programming to overcome common problems in oo software development/evolution", In: Proc. of 21st International Conference on Software Engineering, IEEE Computer Society Press, pp. 687-688
12. Rashid, A., Sawyer, P., Moreira, A., Araujo, J. "Early Aspects: A Model for Aspect-Oriented Requirements Engineering", In: Proc. of IEEE Joint International Conference on Requirements Engineering, pp. 199-202 (2002).
13. Stankovic, J. A. et al., "VEST: An Aspect-Based Composition Tool for Real-Time System", In: Proc. of 9th IEEE RTAS, pp. 58-59 (2003).
14. Stein, D., Hanenberg, S., Unland, R. "Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design", Proc. of 5th Int. Conf. on Aspect-Oriented Software Development, ACM Press, (2006), pp. 15-26.
15. Tsang, S. L., Clarke, S., Baniassad, E. "An Evaluation of Aspect-Oriented Programming for Java-based Real-Time Systems Development", In: Proc. of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04), (2004).
16. Wehrmeister, M.A., Freitas, E.P., Wagner, F.R., Pereira, C.E. "Applying Aspect-Orientation Concepts in the Model-Driven Design of Distributed Embedded Real-Time Systems", To appear in the Proceedings of the 10th IEEE ISORC'07, (2007).
17. Zhang L., Liu, R. "Aspect-Oriented Real-Time System Modeling Method Based on UML". In Proc. 11. IEEE RTAS'05, (2005).

SMART SPEED TECHNOLOGY™

Results of Modeling for Embedded Applications

Mike Olivarez and Brian Beasley
Freescale Semiconductor

Abstract: This document presents the results of using the SpecC Methodology for creating architectures offered by Freescale for the mobile device space, implementing Freescale's Smart Speed Technology. The solutions incorporate a generic processor architecture for code compatibility, but requires additional peripherals to enable the full system level requirements of the solution. A summary of how the methodology has been used to enable the system to contain the needed capabilities shows that a higher level of confidence can be introduced into a complex embedded system at higher levels of abstraction. The overview will give a short explanation of the SpecC Methodology and some examples of how the methodology was used to verify the system for meeting the desired needs for performance, cost, and market requirements, benefiting both the engineering and business aspects of creating a solution.

Key words: Modeling, SpecC, Methodology, Smart Speed, MXC, i.MX, Specification, Architecture, Communication, Energy.

1. INTRODUCTION

The trend toward the use of high level languages such as C/C++ [1, 6, 9] as a basis for the next generation modeling tools and platform methodology to encompass design reuse has come to be a great mechanism. As there is no commercially available proven tool suite to fully implement C to silicon, the key has been to implement the methodology using means available at the current time, in the hope that the tools industry will catch up soon. The Methodology chosen, independent of any tools or language, was the SpecC Methodology. It is a well documented method of going from the C Model of

Olivarez, M., Beasley, B, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 231–240.

the problems, to creating the hardware and software needed to bring the solution to market.

As the problem is much larger and there is a need to avoid proprietary information, only an overview of how the methodology was used to create the architecture will be shown in this document. The document will cover the SpecC Methodology at the high level, and break down some of the pieces to show how the models helped with the implementation of Freescale's wireless and mobile processors, and how finding solutions early in the process allowed for trade-offs to be made based on performance versus cost versus market needs.

This document is organized as follows: In section 2, the SpecC Methodology is explained at a high level. In sections 3, 4, and 5, brief overviews of the different model types and how they impacted the system are explained. Section 6 gives the overall benefit of using the methodology to get products to market. Conclusions are provided in section 7.

2. THE SPECC METHODOLOGY

The SpecC Methodology has uniquely defined models for Specification, Architecture, Communication and Implementation [1, 2, 7]. These models allow the system designer to specify abstracted capabilities based on a C program of what the system needs to accomplish. Progressing through the various models reduces the amount of abstraction until one gets to the implementation details. The Specification, Architecture, and Communications models break the system design tasks down, to be more manageable, yet work toward the common goal. As the system's requirements and definitions increase in complexity, it is much more important to split into parts the tasks using a method that allows them to be divided amongst a team of people to work on, across disciplines and geographies. As the definition of an embedded system becomes blurred from the Application Specific Integrated Circuit (ASIC), which performs a single task, to the fully Asymmetrical Parallel Processing environment such as Freescale's MXC (MX300-30 and MXC275-30 for the sake of this document) [9] line of cellular processors including communications and applications processing, the ability to work more efficiently among various teams is critical to the success of the program.

Utilizing the well defined models allows for different members to work independently on various tasks, yet bring the solution together successfully. The methodology's well defined steps for simplicity allows it to function at the various levels, whether it be at the specification and design of the consumer device and how the various components need to be integrated; or

at the chip level, and how the hardware and software needs to interact and what blocks need to be defined; or at the definition of a specific block and the capability it must contain. Using this top down approach for system level design in a systematic fashion allows for the smooth interaction of the data and helps to create the prioritization of where resources need to be assigned to create a solution. The problems that are not on the critical path can be given a lower prioritization and resources can be quickly identified and aligned using the methodology.

3. SPECIFICATION MODEL

Early definition of the system specification based on requirements and use cases is critical for ensuring the system will meet the needs. As the definition of embedded system is blurred, it becomes increasingly difficult to meet the needs, especially if the system will need to meet future requirements as well. To adjust for this complexity, it is a challenge to understand if designing in additional capabilities will have the effect of being a "value add" to the system, or if it will simply be cost prohibitive to the system. Using the Specification Model and well defined use cases, allows one to make the cost benefit analysis of the system, based on how well it meets the use cases.

The example to be used is the inclusion of the floating point co-processor to the ARM1136J-S™ processor. This allows generic programs to take advantage of floating point capabilities, without having to emulate them in software, thus dramatically improving the performance of algorithms used by applications for audio, graphics and physics, to name a few. At the system level definition stage, it was not known if floating point was a hard requirement, thus, a cost analysis method was needed to determine if the benefit would make the solution cost prohibitive. Using the Specification Model of the SpecC Methodology allowed for the cost analysis to be performed. The Specification Model is a high-level model of the functionality that can test the algorithmic differences, without knowing the implementation details. Timing information can be ignored while cycle approximate capabilities are used to determine the overall difference. This higher level of abstraction allows for more simulations to be run quickly, to enable the system designer to have ample data to make the trade-off analysis. An additional benefit was the ability to estimate performance capabilities of adding parallelism into the system based on the defined use cases, without incurring a large computational increase for running the models. Since the Intellectual Property (IP) for this case is provided by ARM, as well as

simulation tools, the ARM Development Suite (ADS) was used for the basis of these tests.

The setup of the model utilized the Color Space Conversion application used for video. It was broken into behaviors with the top being the application, a second being the I/O, and the other being the floating point processing. There were multiple benefits of using this as a test case. One is that the mathematical algorithm is widely available [8]. A second benefit is that the fixed point code for the function that had been previously verified is also available, so the trade-off between hardware implementation and software implementation can be easily measured. A third benefit is that since the code is small and primarily floating point, the results can easily be multiplied by the percentage of floating point code in a use case to determine a benefit for other use cases by knowing the tendency, without having to fully test other use cases. For instance, since the geometry processing for 3D graphics would also be primarily floating point calculations, it will mimic the benefit shown by this smaller test.

An additional benefit of this kind of test that is an addition to the SpecC Methodology, is that power trade-offs can also be estimated at this point. The power can be estimated using Equation 1 (shown in Figure 1), when the IP blocks to be used are known or can be reasonably estimated. If the needed additional IP has not been specified, the maximum gate count and power specifications can also be determined using this method. The result is the solution can be determined based on the: 1) cost of the IP blocks and rough silicon area needed; 2) performance trade-off of the system as determined from the hardware and software; 3) power utilization of the system and impact on battery life.

From Table 1, we see that the additional hardware for performing floating point calculations, equates to an 825% performance improvement overall, greatly enhancing the floating point performance. Equation 1 shows the energy savings based on the cycles saved (X), and the amount of logic that is added (Y). The energy savings is independent of the technology used and additional techniques within the implementation that may save even more energy, resulting in longer battery life. From all of these factors, one can determine if the additional hardware provides a benefit that would be a "value add" to the overall system, based on the cost of adding the additional capability. Also shown is an alternative fixed point solution, with the results in Table 1. If it is determined that the "value add" is not of a high enough benefit, optimizations in the software can help accomplish a similar benefit, though for floating point intensive applications and dynamic range, the floating point co-processor brings the best benefit.

Of course in business, everyone has an obligation to control cost, and these techniques allow the system designer to do his/her part in the analysis.

The cost for the hardware and software versions can be calculated, based on the size of the code and the size of the additional hardware for the different proposed solutions. At this time, any licensing fees can also be added in and amortized over the project based on the number of units to be sold to obtain the per unit cost. From this, the project can move forward and tasks distributed across resources, or it may be determined that a scrub of the "specification" may need to occur. As this is an iterative process and may benefit multiple projects, the use of the Specification Model and the ease as to which it can be changed, makes this process much easier and more efficient over previous ad hoc methods.

Table 1. Summary of improvements over Software Floating Point.

Improvements over Floating Point Software	Floating Point Hardware Improvement Percentage	Fixed Point Software Improvement Percentage
Application cycles	825%	837%
Floating point calculation cycles	1525%	793%[1]
Instructions issued for floating point calculations	2223%	923%[1]
Application energy savings	83%	88%
Floating point calculation energy savings	91%	88%[1]
Code memory footprint savings	16%	16%
Arithmetic code development time	15 min.	~120 min.

$$PercentEnergySavings = (1 - (\frac{1}{X} * (1 + Y))) * 100\%$$

Figure 1. Power savings, based on X as the cycle reduction multiplier, and Y as the additional logic adder.

4. ARCHITECTURE MODEL

Once the specification has been created, and the behaviors required are being met, the next step is to decide the best way to implement. Behaviors such as adding a floating point unit or not are still a high-level trade-off, but they're made easy if the decision is whether or not to add the capability. The architecture model goes into what is the most efficient way to implement the behaviors that have been defined. Again, the analysis can include not only the engineering aspects of architectural exploration, but also the cost, die size, and time to market aspects.

[1] Results are not just isolated arithmetic, but also contain I/O requirements for use of the data structures, as opposed to the calculations used for floating-point arithmetic.

Within the architectural exploration process of refining the specification, trade-offs of the System on-chip (SoC) IP blocks can be measured for the performance impact of hardware versus software for implementing the behavior, as well as how the various blocks are connected within the system. Various IP blocks that implement the behavior can be tested, as well as various hardware/software partitions. In many cases, reuse is most important for backward compatibility and time to market, and specific IP will be taken off the shelf. In other cases, new IP can be introduced with different hardware and software trade-offs. In either case, modeling the system will ensure the behaviors are functioning such that the system specification is met [3]. Performance of the behavior can still be quite different in how the SoC is architected, even with the same IP blocks being used. The trade-off analysis to be used for the example in this section is the placement of the level 2 (L2) cache in the system.

Architectural exploration was performed to see if the L2 cache subsystem should be dedicated to the ARM processor, or if it should be shared across multiple processing elements (PE) within the system. The characteristics for the memory subsystem were set up, based on the times required for data to be obtained from the level 1 (L1, instruction and data, I-cache and D-cache respectively) cache of the CPU and level 2 cache (L2) accesses. The statistical model used 30% of the instructions as memory access (10% stores, 20% loads overall). The I-cache was set to a 98% hit rate, and the D-cache was set to a 95% hit rate, while the L2 utilized a 50% hit rate and other blocks were given statistical access rates to complete the model. The memory arbitration scheme was done using priority arbitration, where the priority was set, based on the processing element number, with the CPU as 1 and the others sequentially added up to 5 elements in the model.

The simulation looked at the cycles per instruction (CPI) the ARM processor should be able to achieve, as well as the theoretical CPI (tCPI) of the SoC, based on each of the PEs having different definitions of instructions. Since the two architectures are utilizing the same types of PEs, the measurement is a good comparison of the two architectures. If the two systems will be utilizing different PEs, the effective CPI (eCPI) could be calculated based on simulations of test cases as defined in [4]. Due to the difference in the speed of the CPU and its higher memory bandwidth needs compared to the other PEs used in the test, having the L2 dedicated to the CPU gave the best performance, as shown by the lower CPI numbers, for both the CPU and the SoC tCPI[2]. The test was also run with a faster memory interface, as could be accomplished using double data rate memory and shared L2, which improved on the numbers, but still wasn't as efficient as

[2] tCPI based on statistical simulations and is an approximate measure of what the architecture may achieve. eCPI is based on measured test cases using actual applications.

what was accomplished using the architecture when the L2 cache is dedicated to the ARM processor.

Using the Architecture Model, it was shown that the L2 cache was a greater benefit to the much faster processing element. A secondary result was the number of external memory access was reduced by allowing all the PEs to utilize the on-chip memory. Lowering the overall number of off-chip memory access could be an energy savings, but the difference was not shown to be great enough to warrant the decrease in performance for this case. In a separate case where the dominate PE was not significantly faster than the other PEs, it was shown that sharing the on-chip cache memory was more beneficial for the overall performance and lowered the number of off-chip memory accesses, thus conserving precious system energy. Even though the PEs may not change, the subtle changes in the behaviors defined through architecture exploration can have a significant impact on the overall capabilities of the system.

The Architecture Model allows one to quickly make trade-offs between different configurations, implementing the behaviors defined by the Specification Model. Quickly changing between scheduling and priorities allows the system designer to make architectural decisions, while abstracting out information such as the communication protocol and overall bus structure. This method can be used hierarchically so that the problem is more manageable and can be split between different architects and teams.

5. COMMUNICATIONS MODEL

The Communications Model allows less abstracted system timing to be introduced to get a true performance estimation of the behaviors. The Specification and Architecture models can only use a rough estimation of the timing, and didn't include any overhead that the bus structures would introduce. When including the various IP blocks of the system, one must also take into account the interface portion, as the various IP vendors may not use the same bus interface for all the blocks. The Communications Model enables this to be taken into account.

As the various PEs are added to the system, the arbitration schemes and bus structures have a large performance impact, as well as cost and time to market implications. The easiest method is to connect every processing element (including the memories) to a single bus. The bus will need to arbitrate between all the elements, and will only be as fast as the slowest element. The best performance would have to be a mesh network of busses, so that every processing element has a point-to-point connection with every

other element at the rate of the fastest element. This latter method is too expensive, and wastes much of the resources added to the system.

In the MXC and i.MX products, various bus structures are utilized to maximize the performance and cost. For the high performance processing elements and the memory interface, a Smart Speed Switch has been introduced, to allow the faster processing elements to communicate on a point-to-point basis, with fast access to memory. These processing elements have a standardized bus interface definition, to ensure ease of connectivity between standard IP blocks with the highest performance. The slower peripherals use a slower, more cost-effective bus and are interfaced to the shared resource. Such a method allows for the slower devices to be efficiently integrated into the system, without impacting the performance of the more timing- and bandwidth-critical components.

Since the two domains still need to be able to share resources such as memory, and allow inter-process communication, transducers were added to the system to close the gaps. The two primary standards used in this case are ARM's AMBA AHB bus protocol for the high speed communications, and the Freescale Intellectual Property Interface (IPI) structures for the less performance-critical elements.[3] Bridges have been created between the different bus architectures, and modeled for performance using an internal modeling tool based on SystemC [6]. A result of the modeling was the creation of the Smart Speed Switch, which is an AHB-compliant crossbar that allows up to five master elements to talk to slave elements simultaneously, giving the relative ability to enable data throughput of a 665 MHz bus.

A key design parameter the system will need to meet for the mobile space is to efficiently complete use cases without wasting resources. The best method for lowering clock speed and saving precious battery life is to utilize parallelism in the system, and using dedicated processing elements, as was shown previously with the floating point example. Looking at the complex behaviors of the system, parallelism will need to exist not only by the various PEs working simultaneously, but also in the communications, such that the PEs are not starved. By using the Communications Model and adding in the requirements for the protocol and bridging between protocols, one can determine various bus configurations for the system. Through such modeling of the use cases and deciding on the PEs for the architecture, it was determined that there was sufficiently enough high bandwidth or low latency peripherals to warrant a switch which allows for up to five masters. Such PEs can be represented by the ARM, L2 Cache, Image Processing Unit,

[3] The Freescale IPI standard includes a range of bus structures for different speed peripherals. The slower speed bus is used, as it connects the non-speed critical processing elements for the system mentioned in this document.

Multi-channel DMA Unit, Memory Controller, High Speed I/O, Graphics Processor, and bridges to lower bandwidth/latency PEs, though all of these may not be a master. Complex use cases can be used to best define the communications requirements for such systems.

Of course, the parallelism doesn't come for free. There are costs of silicon space and energy consumption to be factored. Equation 1 can be used once again to estimate the costs over a previously utilized solution. The ability to quickly configure and change the model allows for the communications structures to be changed to meet the desired cost targets, before getting too deep into implementation details.

6. CREATING SMART SPEED TECHNOLOGY

Smart Speed Technology is the brand that defines a system solution, based on providing enough performance for the use cases, without wasting resources. Unlike the personal computing space, where processors are run as fast as possible, even if the use case defines an idle state, thus wasting resources, processors for the mobile space must be able to conserve as much resources as possible, without degrading the user experience. Keys to accomplish this goal are to begin with well-defined use cases that create the foundation of the solution, and to add some use cases that are good candidates for future requirements. From the use cases, the Specification Model can be built, tested and verified. As capabilities are added that could drain the precious limited energy resource of the battery, additional capabilities to limit power consumption can be added. In the case of MXC and i.MX processors, additional power saving capabilities such as Dynamic Voltage and Frequency Scaling and Dynamic Process Temperature Compensation have been added to conserve energy.

Benchmarking can be run on the final solution, to verify the methodology and improve upon it. Via the use of published data from Synchromesh Computing's benchmarking of the i.MX31 processor [10, 11], it can be seen that the modeling of the system made for accurate trade-off and specification analysis. Though the difference is 17% greater efficiency by the final solution over what was projected for floating point tests, it can be seen that the data is "Close enough for all practical purposes" [5] to make the decisions at a high level of abstraction. This affirms the use of the Specification Model as a tool for making high-level system trade-offs. Additional benchmarks based on Consumermark™ and STREAM benchmarks shows the architecture can efficiently perform the use cases they test, verifying the use of the L2 Cache and overall system communications. The benchmarking also proved the power savings the architecture was

estimated to obtain. Based on the benchmarks, it can be seen that the overall goals of the system for performance and cost and the overall price/ performance targets specified have been met.

7. CONCLUSION

It can be seen from the techniques shown, as well as in the benchmark results of the final product, that the modeling effort used to design the architecture for the MXC and i.MX products allowed for Smart Speed Technology to be highly leveraged. The solutions can meet the use cases computationally, yet still achieve low clock speeds and energy efficiency for the mobile system. This achievement can be attributed directly to the SpecC Methodology for performance with additional analysis for power and cost to make a solution that is successful in the market for mobile consumer devices.

REFERENCES

[1] D. Gajski, J. Zhu et al. "SpecC: Specification Language and Design Methodology" Kluwer Academic Publishers, 2000.
[2] www.ics.uci.edu/~specc
[3] L. Cai, M. Olivarez, D. Gajski, "Introduction of System Level Architecture Exploration Using the SpecC Methodology", International Symposium on Systems and Circuits, May 2001.
[4] M. Olivarez, B. Beasley, "Use Effective Cycles/Instruction as a True CPU Benchmark", Portable Design, May 2005.
[5] S. Nichols, "Rules for Engineering", Various Lectures: University of Texas at Austin, 1987-.
[6] http://www.systemc.org
[7] A. Gerstlauer, R. Dömer, D. Gajski, "System Design: a Practical Guide of with SpecC", Kluwer Academic Publishers 2001.
[8] V. Bhaskaran and K. Konstantinides, "Image and Video Compression Standards: Algorithms and Architectures, Second Edition", Kluwer Academic Publishers, 1997.
[9] http://www.freescale.com/smartspeed
[10] http://www.eembc.com
[11] "The Freescale Semiconductor i.MX31 Processor: Cool, Powerful", Synchromesh Computing, 2005.

Smart Speed, MXC, and i.MX are marks of Freescale Semiconductor.
ARM, ARM1136J-S, ARM1136JF-S, and ARM Development Suite are trademarks of ARM Ltd.
Consumermark™ is a trademark of EEMBC.

POWER OPTIMIZATION FOR EMBEDDED SYSTEM IDLE TIME IN THE PRESENCE OF PERIODIC INTERRUPT SERVICES

Gang Zeng, Hiroyuki Tomiyama, and Hiroaki Takada
Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan

Abstract: Generally, there are periodic interrupt services in the real-time embedded systems even when the system is in the idle state such as the periodic clock tick interrupts. To minimize the idle power, power management therefore should consider the effect of periodic interrupt services. In this paper, we deal with this problem considering two scenarios. In case the periodic interrupt cannot be disabled, we first model the power consumption and then propose static and dynamic approaches for the optimal frequency selection to save idle power. On the other hand, in case the periodic interrupt can be disabled, we propose an approach to delay the interrupt service until the next task is released so that the processor can stay in low power mode for longer time. The proposed approaches are implemented in a real-time OS and its effectiveness has been validated by theoretical calculations and actually measurements on an embedded processor.

Key words: dynamic power management; dynamic voltage/frequency scaling; real-time embedded systems

1. INTRODUCTION

Energy consumption has become one of the major concerns in today's embedded system design especially for battery-powered devices. For the sake of dependability, in real-time systems the utilization of processor is less than 100% even if all tasks run at WCET (worse case execution time). Moreover, workload of each task may vary from time to time, which results in the less average execution time than the WCET. All these factors lead to the system idle state in which there are no tasks needed to be scheduled. It

Zeng, G., Tomiyama, H., Takada, H, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 241–254.

should be noted that even in the idle state, most real-time OS maintains a periodic clock interrupt to synchronize the system and trace the clock events. For example the uc/OS-II, eCOS, and Linux need a 10ms clock interrupt to generate the system clock. Besides the period clock tick, some interrupt-driven embedded systems such as data acquisition systems also need periodic interrupts to activate the CPU from low power mode for data processing. To reduce the power of the idle state, a common approach is to transfer the processor into a low power mode. Generally, a processor can provide multiple low power modes to deal with different system states. To take advantages of these power control mechanisms, dynamic power management (DPM) tries to assign the optimal low power mode according to the predicted duration of the system idle state. As an example, Figure 1 shows the power mode transition graph for two typical embedded processors in high-end and low-end applications, respectively. Two observations can be derived from Fig. 1 as follows: (1) Although different processors may have different names of low power modes, they utilize similar techniques for power control by disabling either CPU clock or both CPU and peripheral clocks. (2) The power mode transitions consume both time and power overhead which are dependent on the specified low power mode and the complexity of processors.

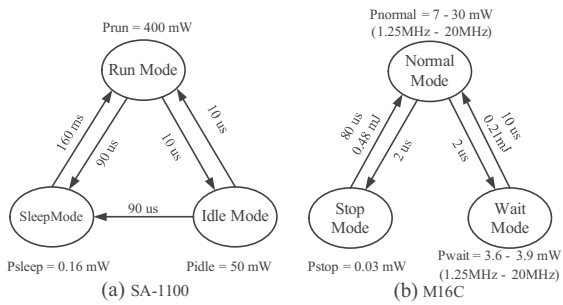


Figure 1. Power mode transition for (a) Intel's StrongARM SA-1100 processor [1] (b) Renesas's M16C processor.

While the SA-1100 with integrated 32-bit RISC core targets for high performance low power application, the M16C [11] with integrated 16-bit CISC core, on-chip ROM and RAM aims at low-end and low power application. The SA-1100 processor provides three operation modes with different power consumption levels, i.e., Run, Idle, and Sleep modes. The Run mode is the normal operating mode with full functionalities and high power consumption. In contrast, the Idle and Sleep modes are low power modes with stopped CPU clock. Idle mode stops the CPU core clock but enables all peripherals clock thus on- or off-chip interrupt service requests

can quickly reactivate the CPU. On the other hand, Sleep mode stops both CPU and peripherals clock thus only hardware reset or special event can wakeup the CPU, which requires long transition time whenever entering or exiting the Sleep mode. Similarly, the M16C also provides three power modes which have similar functionalities to that of the SA-1100 but with different names. However, the time and power overhead of M16C for power mode transition is much less than that of SA-1100. This small transition overhead of M16C is benefited from its simple and single-chip architecture. Actually, only one instruction is needed to transfer the processor into wait mode.

Although the Sleep mode of SA-1100 has the lowest power consumption, it is not suitable for the application considered in this paper. The reasons are that (1) the transition time overhead for returning to run mode is too large to be used in the application with short period of interrupt services; (2) the normal interrupt service requests related to on-chip clock cannot work properly in the Sleep mode. Therefore, the feasible low power mode that can be used for power management of idle time with periodic interrupt services is the idle mode.

In addition to DPM, another effective technique for power saving is dynamic voltage/frequency scaling (DVFS), because the power consumption of CMOS circuits is proportional to its clock frequency and its voltage square. The DVFS tries to change the clock frequency and its corresponding supply voltage dynamically to the lowest possible level while meeting the task's deadline constraint. Commonly, the voltage and frequency scaling are accomplished by controlling a DC-DC converter and PLL (phase lock loop) circuit, respectively. Although many high-end processors have equipped with the DVFS capabilities, few low-end processors can dynamically change their supply voltages such as the M16C. In contrast, most low-end processors can still change its clock frequency by setting the divider registers. As a result, the time overhead for frequency change is much less for a simple processor using divider register than a complex processor using PLL. For example, the M16C requires negligible time for frequency change. In contrast, many commercial high-performance processors require the transition time ranging from 189us to 3.3ms for voltage and frequency scaling [10]. For simplicity, we refer to DVFS in the following whenever voltage and frequency or only frequency is changed during execution.

The motivation for this work stems from the fact that the power consumption of processor in idle mode is not fixed but dependent on the selected clock frequency before entering the idle mode [7]. In general, the higher frequency, the more power is consumed in idle mode. For example, the PXA225 processor (an upgraded product of SA-1100 series) consumes 45mW-121mW power in idle mode which corresponds to 100MHz -

400MHz frequency, respectively [7]. The reason is that although the disabled CPU cannot consume dynamic power in idle mode, the enabled peripherals still consume power which is directly dependent on the selected clock frequency [8]. To reduce the power of idle mode we therefore expect to lower the frequency of processor. However the lowered frequency will lead to longer execution time for interrupt service routine (ISR), which may result in higher total energy. Accordingly, we need to determine the optimal frequency for the idle state with period interrupt services to save power. To the best of our knowledge, this is the first work that addresses the problem of selecting the optimal frequency to save power for idle time in the presence of periodic interrupt services. The main contributions of this work are as follows: (1) In case the periodic interrupt cannot be disabled such as the data acquisition systems, we first model the power consumption and then propose static and dynamic approaches to save idle power for the processors with large or negligible DVFS overhead, respectively. (2) In case the periodic interrupt can be disabled such as the clock tick interrupt, we propose configurable clock tick to save idle power and keep system time synchronization.

The rest of the paper is organized as follows. Section 2 gives related work. Section 3 presents the power model and the proposed approaches. In Section 4, experimental results are described. Finally, Section 5 summarizes the paper.

2. RELATED WORK

Recently, there have been a large number of publications using DPM or DVFS for power savings. Most DPM literatures focus on the design of power management policies using predictive schemes or stochastic optimum control schemes [1,4]. In these schemes, they generally assume fixed power consumption for each low power mode and their objective is to decide when and which low power mode the devices should transfer into. In practice, an on-chip timer interrupt is commonly employed in embedded systems to reactivate the CPU from low power mode quickly. In this case, the on-chip clock cannot be disabled, which results in varied power consumption in low power mode as mentioned in Section 1.

While DPM aims to reduce power in the long idle time by transferring the processor into low power mode; DVFS aims to save power in the short slack time, which is generated due to the fluctuation of workload, by lowering the processor voltage and/or frequency. Most DVFS algorithms assume periodic tasks with known WCET and deadline. Although the objective of DVFS is to prolong the task execution time until deadline by

lowering the CPU's voltage and frequency, the slack time cannot be reclaimed completely. This is because the generated slack time can only be reclaimed when there are ready tasks that can be scheduled immediately. Moreover, the discrete frequency levels makes DVFS cannot utilize the generated slack time completely. All these factors result in idle time even in the DVFS enabled systems. However, most DVFS literatures ignore the idle time process by simply assuming a low power mode with zero power [2,3] or fixed power consumption [10] in idle time. As shown in Section 1, even in low power mode, the power consumption is neither zero nor fixed value, which is dependent on the on-chip clock frequency. Moreover the required time for power mode transition may be too long to be applicable for short idle time, which results in no power reduction in this case.

Recently, a variable scheduling timeouts method is proposed for power savings in Linux systems by eliminating the useless tick interrupts during system idle time [9]. However a problem needed to be considered in real-time systems is how to keep the system clock synchronization caused by tick timer reprogramming.

3. POWER MODEL AND APPROACHES

For general low power embedded processors, we assume that the processor can provide multiple low power modes and alterable voltage/ frequency for power control. To simply the calculation, we assume that the time and power overhead for power mode transition and voltage/ frequency scaling are fixed. As discussed earlier, for power management of idle state with periodic interrupt services, only the low power mode with enabled peripherals clock is considered. We assume that an idle task is employed to implement the proposed power management in RTOS. The idle task is scheduled to run when system enters the idle state in wnich no tasks need to be scheduled in the ready queue.

We deal with the power saving problem of idle state in two different cases. While in case one the periodic interrupt cannot be disabled such as the data acquisition system, in case two the interrupt can be disabled for a specified duration such as the clock tick interrupt.

3.1 Case one: the periodic interrupt cannot be disabled

Before modeling the power consumption of idle state with periodic interrupt services, we give the following notations.

- M: selected system speed, i.e., 1/M full speed
- T_p (us): period of interrupt service

- T_h (us): execution time of interrupt service routine at full speed
- T_s (us): execution time for low power mode setting in idle task at full speed
- T_p (us): time overhead for power mode transition
- I_p (mA): average current during power mode transition
- T_v (us): time overhead for dynamic voltage/frequency scaling
- I_v (mA): average current during voltage/frequency scaling
- I_{rm} (mA): the run mode average current at 1/M full speed
- I_{im} (mA): the idle mode average current at 1/M full speed
- V_m (V): the corresponding voltage for 1/M full speed setting

Considering the fact that different scale processors may have different DVFS overhead as discussed in Section 1, we propose static and dynamic approaches for processors with large or negligible DVFS overhead, respectively.

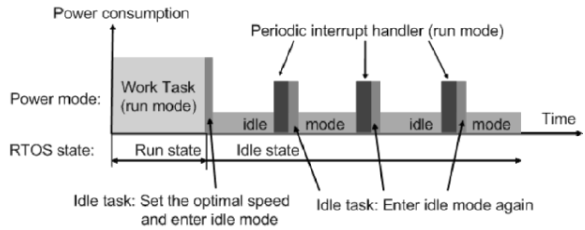


Figure 2. Processing procedure for idle state power management.

If the processor has large DVFS time overhead, a static approach is adopted, i.e., only once DVFS setting at the beginning of idle state for any continuous idle time. Specifically, the program in idle task takes corresponding actions according to the current system state, if it is the first time to enter idle state, it first sets the optimal speed for power savings and then enters low power mode. Otherwise, it only sets and enters the low power mode and without any speed change when the idle task is reactivated from low power mode by interrupt. The above processing procedure for idle power management is illustrated in Fig. 2. Based on the above notations and procedure, the average current and power of idle state with periodic interrupt services can be calculated by the following equations:

$$I_{idle} = \frac{(T_h + T_s) \cdot M \cdot I_{rm} + (T_p - (T_h + T_s) \cdot M - T_p) \cdot I_{im} + T_p \cdot I_p}{T_p} \quad (1)$$

$$Power_{idle} = I_{idle} \cdot V_m \quad (2)$$

Therefore, if the period of interrupt and the execution time for power mode setting are known and fixed, time and power overhead for power mode transition, the average current with different speed settings for run and idle mode can be obtained from processor data manual or actual measurements, the average current of idle state will be a function of the selected speed M and the execution time of interrupt service T_h. According to this function, the power optimization problem can be formulated as: for a specified processor and application with known T_h, T_s, T_p, I_p, I_{rm}, and I_{im}, finds the optimal M such that the average idle current is minimal. Because the relation between I_{idle} and M is linear, and the selectable speeds are limited, we can calculate all curves of I_{idle}-T_h with all possible speed selections, and then the one that has the minimal average current will be the optimal speed setting.

If the processor has negligible DVFS time overhead, a dynamic approach may save more power at the expense of two DVFS settings for each interrupt process. The procedure is that the full speed is set at the beginning of each interrupt service, and the slowest speed is set before entering the low power mode each time. Its objective is to save more power by keeping the processor in low power mode with the minimal power consumption for longer time. In this case, the average idle current can be calculated by the following equation:

$$I_{idle} = \frac{(T_h + T_s) \cdot I_{r1} + (T_p - (T_h + T_s) - T_t - 2T_v) \cdot I_{in} + T_p \cdot I_p + 2T_v \cdot I_v}{T_p} \quad (3)$$

where I_{r1} represents the current of full speed running, and I_{in} represents the current of the slowest speed in idle mode. Note that this approach is not realistic for some complex processors with large DVFS overhead. For example, Intel's PXA225 requires 500us for each DVFS scaling [7]; obviously in this case, the dynamic approach is not applicable for the interrupt service with 1 ms period.

3.2 Case two: the periodic interrupt can be disabled for a specified duration

We assume periodic tasks with known WCET and deadline in embedded systems, and we only discuss how to disable clock tick interrupt by using a configurable clock tick in order to save more power during idle state. Under the above assumptions, whenever system detects the beginning of an idle state, it also knows the nearest releasing time of a periodic task. In this case, the duration of the idle state is known, we therefore can disable the clock tick for this known idle time and transfer the processor into low power mode to save power. Note that this approach is different from general DPM in that

while general DPM makes decision for power mode transition based on the predicted duration of idle time; this approach is with known duration of idle time. Therefore the decision for power mode transition in this approach is straightforward.

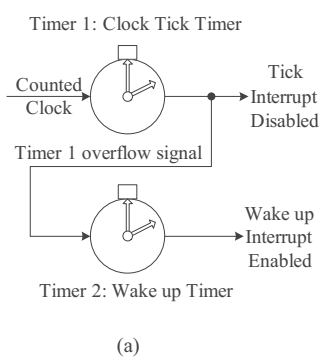


Figure 3. Configurable clock tick and timer setting procedure.

When the clock tick interrupt is disabled during idle state, a problem that should be considered is how to trace the original clock tick to keep system time synchronization. To this end, another timer, as shown in Fig. 3, can be used to count the lost ticks during idle time when the tick interrupt is disabled. Because the original tick timer is never stopped and restarted except disabling its interrupt requests, the system time synchronization can be guaranteed easily. However, this approach is hardware-dependent since a wire connection between the output of timer 1 and the input of timer 2 is required as shown in Fig. 3 (a). The count value of timer 2 for generating the wakeup interrupt prior to the release of next task should be set to the known duration of idle state. The detailed timer setting procedure is listed in Fig. 3 (b). In conjunction with the configurable clock tick, the complete algorithm for idle time power management is given in Fig. 4.

Algorithm for idle time power management:
(assume the time overhead for power mode transition is larger than that for DVFS)
When system enters the idle state
1. Calculate the duration of idle time
2. **If** (the duration > the time overhead for power mode transition) **then**
3. enable and set the configurable clock tick as the Fig. 3
4. set the slowest speed
5. enter the low power idle mode
6. **else if** (the duration > the time overhead for DVFS)
7. set the slowest speed
8. **end if**

Figure 4. Power management algorithm when the periodic interrupt can be disabled.

4. EVALUATION AND EXPERIMENTAL RESULTS

4.1 Experiment setup and measurement environment

To validate and evaluate the proposed approach, we select the OAKS16-mini board with a M16C (M30262F8GP) embedded processor to implement the approach. Although the processor cannot change its supply voltage, it provides three power modes and can quickly change its clock frequencies by setting the divider registers. We measure the processor current by inserting a digital multimeter between the power supply and the power pin of the processor. An oscilloscope is utilized to observe the voltage waveform of the shunt resistor which is inserted between the power supply and the power pin of the processor. The time and power overhead for power mode transition are estimated by using the captured voltage waveform. Note that the above experiments are performed separately so that the current measurements are carried out with removed shunt resistor. The measured power results and estimated power mode transition overhead are given in Fig.1.

Our approach has been implemented in a RTOS called TOPPERS/JSP kernel [5] which is an open source RTOS in consistent with the ITRON [6] standard. The TOPPERS RTOS targets for real-time applications with limited resource requirement. A configurable clock tick is implemented in OS with default 1 ms interrupt period. The normal execution time of the timer handler for system time updating is about 12 us at 20MHz.

4.2 Evaluation of the proposed approach when the periodic interrupts cannot be disabled

Table. 1 summaries the measured normal and wait mode average current under different speed settings. Note that all these measurements are performed by executing a busy loop and the results for wait mode is measured with clock enable but without any interrupt services.

Based on these measured parameters, and Eq. (1), we can obtain the following current vs. execution time and speed curves in Fig. 5. From this figure, it is clear that the optimal speed selection for minimal power consumption is determined by the execution time of ISR. As for the 12us interrupt service in this experiment, the optimal speed is 10MHz (1/2 full speed). The calculated and measured results are denoted in Table 2, respectively, where the minimal measured current is consistent with the theoretical calculated results. We modify the ISR and reduce the execution time of ISR to 7us, and perform the above experiments again. As can be seen from the results given in Table 3, 5MHz (1/4 full speed) can achieve the

minimal power consumption, which is also consistent with the calculated results.

Table 1. Measured normal and wait mode average current under different speed settings

Selectable Speeds (1/M full speed)	Measured current (mA) (voltage = 3V)	
	Normal mode: I_{rm}	Wait mode: I_{im}
20MHz (1/1)	10.04	1.30
10MHz (1/2)	6.35	1.26
5MHz (1/4)	4.35	1.24
2.5MHz (1/8)	3.24	1.23
1.25MHz (1/16)	2.45	1.22

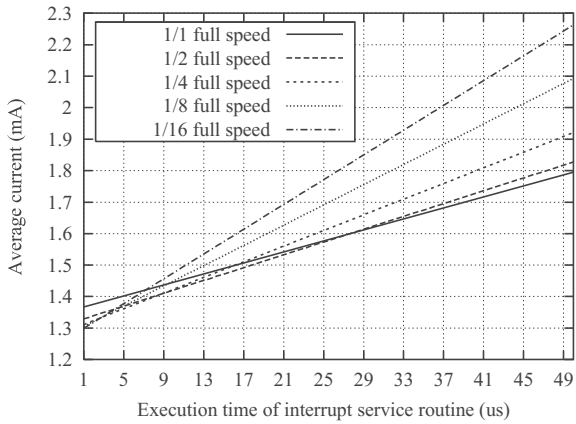


Figure 5. Calculated results for 1ms interrupt period: average current vs. execution time and speed selection.

Table 2. Comparison of measured and calculated average current with Tp=1ms Th=12us

Selected Speed (1/M full speed)	Idle state average current (mA) under periodic interrupt service (voltage=3V, period=1ms, Th=12us)	
	Measured current	Calculated current
20MHz (1/1)	1.47	1.472
10MHz (1/2)	**1.45**	**1.451**
5MHz (1/4)	1.47	1.461
2.5MHz (1/8)	1.50	1.498
1.25MHz (1/16)	1.57	1.534

We change the interrupt period to 10ms and perform the above calculations and measurements again. The corresponding results are given in Table 4. As can be seen, the optimal speed is 1.25MHz (1/16full speed).

When we further prolong the interrupt period to 100ms, the results show that the slowest speed will achieve the minimal power consumption in spite of the variation of execution time. The reason is that for longer interrupt period, most of time the processor stays in low power mode, thus, the average power is dominated by the power of long idle state but not the power of short execution state.

Experiments are also conducted to validate the proposed dynamic approach especially for the M16C with negligible DVFS overhead. In these experiments, the varied speeds are set at the beginning of ISR, and the slowest speed (1/16 full speed) is set in the idle task before entering the low power mode. The calculated results using Eq. (3) and assuming negligible DVFS overhead are depicted in Fig. 6 where the curves for static and dynamic approaches are shown, respectively. As can be seen, the full speed setting for ISR plus the slowest speed setting (1/16) for low power mode outperforms other speed combinations in dynamic approach, and all speed settings in static approach. Meanwhile, the actually measured result for this case shows average current 1.39 mA which is the minimal current compared with the measured results for static approach in Table 2. The results indicate that the dynamic approach can further reduce the average power by 4.3% than the optimal static approach, and achieves the maximal 11% reduction in average power than the approach without frequency selection for idle state.

Table 3. Comparison of measured and calculated average current with Tp=1ms Th=7us

Selected Speed (1/M full speed)	Idle state average current (mA) under periodic interrupt service (voltage = 3V, period = 1ms, Th = 7us)	
	Measured current	Calculated current
20MHz (1/1)	1.40	1.419
10MHz (1/2)	1.38	1.389
5MHz (1/4)	**1.37**	**1.385**
2.5MHz (1/8)	1.38	1.401
1.25MHz (1/16)	1.42	1.416

Table 4. Comparison of measured and calculated average current with Tp=10ms Th=12us

Selected Speed (1/M full speed)	Idle state average current (mA) under periodic interrupt service (voltage=3V, period=10ms, Th=12us)	
	Measured current	Calculated current
20MHz (1/1)	1.32	1.317
10MHz (1/2)	1.28	1.279
5MHz (1/4)	1.25	1.262
2.5MHz (1/8)	1.24	1.256
1.25MHz (1/16)	**1.24**	**1.251**

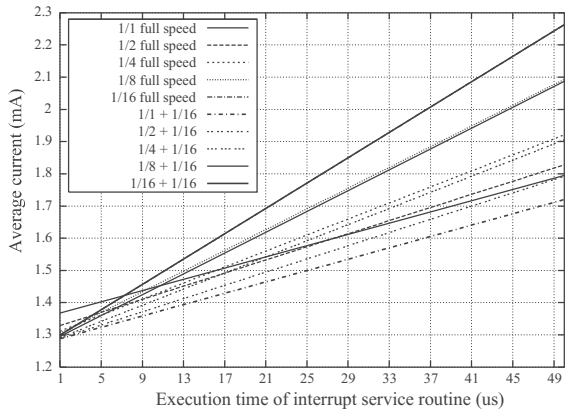


Figure 6. Static approach vs. dynamic approach (with 1ms interrupt period).

4.3 Evaluation of the proposed approach when clock tick interrupts can be disabled

To evaluate the proposed configurable clock tick and idle power management algorithm, Pillai and Shin's DVFS scheduling algorithm called "Cycle-conserving DVS for EDF scheduler" [2] in conjunction with the proposed DPM algorithm are implemented in the TOPPERS/JSP kernel. The experiment test set is presented in Table 5, and corresponding energy results for one minute running are summarized in Table 6. As can be seen, while DVFS can achieve significant power savings compared with full speed running, the proposed configurable clock tick for idle state power management can further reduce the energy by 23% in average compared with normal DVFS without any idle state processing.

Experiment is also conducted to verify the capability of keeping system time synchronization. We implement the configurable clock tick and the original clock tick in RTOS, respectively, and then let them run the above DVFS experiments for 30 minutes. Finally, we compare their system time after running. The results show no difference between the two implementations, which indicates the configurable clock tick can trace the original clock tick precisely even if the clock tick is disable during idle time.

Table 5. Experiment task set

Task set	Period (ms)	WCET (ms)	Actual ET (ms)
Task 1	500-2000	130	28-130
Task 2	500-3000	245	38-245

Table 6. Evaluation of power savings for combined DVFS and power management of idle state

Dynamic EDF DVFS with different idle state process	P1: 500 ms P2: 500 ms	P1: 500 ms P2: 900 ms	P1: 1000 ms P2: 1500 ms	P1: 2000 ms P2: 3000 ms
No DVFS (full speed)	TE: 1807 mJ NR: 1	TE: 1807 mJ NR: 1	TE: 1807 mJ NR: 1	TE: 1807 mJ NR: 1
DVFS without idle state process	TE: 1594 mJ NR: 0.88	TE: 1288 mJ NR: 0.71	TE: 897 mJ NR: 0.50	TE: 468 mJ NR: 0.26
DVFS setting the lowest speed and entering wait mode	TE: 944 mJ NR: 0.52	TE: 773 mJ NR: 0.43	TE: 553 mJ NR: 0.31	TE: 282 mJ NR: 0.16

Note: P: task period (ms); TE: Total Energy (mJ); NR: Normalized result

5. CONCLUSION

Even in a DVFS enabled embedded system, there must exist idle time. Moreover, a periodic interrupt services may be required in the system idle time. As a common approach, the processor can be transferred into the low power mode during idle time, its power consumption however is neither zero nor fixed which is dependent on the selected clock frequency. In this work we present different approaches for idle time power management in the presence of periodic interrupt services. In case the periodic interrupt cannot be disabled, we model the power consumption and propose static and dynamic methods to achieve minimal power consumption for the processors with large or negligible DVFS overhead, respectively. In case the periodic interrupt can be disabled such as the periodic clock tick interrupt, we proposed a configurable clock tick to save power by keeping the processor in low power mode for longer time. We implement the proposed approaches in a RTOS and a frequency scaleable embedded processor. The measured results show that the maximal 11% power can be reduced in the first case, and average 23% power can be further reduced in the second case compared with DVFS without any idle processing.

ACKNOWLEDGMENTS

This work is supported by the Core Research for Evolutional Science and Technology (CREST) project from Japan Science and Technology Agency.

REFERENCES

1. L. Benini, A. Bogliolo, and G. D. Micheli, A survey of design techniques for system-level dynamic power management, *IEEE Trans. on Very Large Scale Integration Systems (VLSI)*, Vol. 8, No. 3, pp. 299-316, June 2000.
2. P. Pillai and K. G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, *Proc. ACM Symposium Operating Systems Principles*, pp.89-102, 2001.
3. W. Kim, D. Shin, H. Yun, J. Kim, and S. L. Min, Performance comparison of dynamic voltage scaling algorithms for hard real-time systems, *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 219-228, 2002.
4. Z. Ren, B. H. Krogh, and R. Marculescu, Hierarchical adaptive dynamic power management, *IEEE Trans. on Computers*, Vol. 54, No. 4, pp. 409-420, April 2005.
5. TOPPERS Project; http://www.toppers.jp/
6. ITRON Project; http://www.sakamura-lab.org /TRON/ITRON/
7. Intel, Application Note, PXA255 and PXA26x applications processors power consumption during power-up, sleep, and idle, April, 2003.
8. Texas Instruments, Application Report, SPRA164, Calculation of TMS320LC54x power dissipation, June 1997.
9. Variable scheduling timeouts (VST) project page; http://tree.celinuxforum.org/ CelfPubWiki/VariableSchedulingTimeouts
10. D. Shin and J. Kim, Intra-task voltage scheduling on DVS-enabled hard real-time systems, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24, No. 10, pp. 1530-1549, Oct. 2005.
11. Renesas Corp; http://www.renesas.com/fmwk.jsp?cnt=m16c_family_landing.jsp&fp= /products/ mpumcu/m16c_family/

REDUCING THE CODE SIZE OF RETIMED SOFTWARE LOOPS UNDER TIMING AND RESOURCE CONSTRAINTS

Noureddine Chabini[1] and Wayne Wolf [2]
[1]*Department of Electrical and Computer Engineering, Royal Military College of Canada P.B. 17000, Station Forces, Kingston, ON, Canada, K7K 7B4, Email: chabini@rmc.ca;*
[2]*Department of Electrical Engineering, Princeton University, Eng. Quadrangle, Olden Street, Princeton, NJ, USA, 08544, Email: wolf@princeton.edu*

Abstract: Retiming has been originally proposed as an optimization technique for clocked sequential digital circuits. It has been successfully applied for optimizing loops during the compilation of loop-intensive programs. After applying a retiming, the original loop transforms to another loop which is preceded by a segment of code called *prologue* and is followed by a segment of code called *epilogue*. To optimize a loop, there are many possible retimings that allow to achieve the same value of the objective function. Depending on the retiming used, the number of operations in the *prologue* and *epilogue* can increase or decrease. Decreasing the code size for retimed loops is of great importance in particular for memory-constrained system-on-chip and embedded systems. It has also an impact on power dissipation. This paper addresses the problem of reducing the code size for retimed software loops under timing and resource constraints. We mathematically formulate this problem and develop algorithms to optimally solve it. Experimental results are also provided.

Key words: Code Size; Loops; Retiming; Timing; Resource; Embedded Systems; Power.

1. INTRODUCTION

While it was proposed originally for optimizing synchronous sequential digital circuits, retiming[9] has been successfully used for optimizing loops as well[1,7,8]. Retiming moves registers to achieve a certain objective function.

There is a relationship between a register in the context of hardware implementations and an iteration in the context of software implementations. In the context of software implementations, moving for instance one register

Chabini, M., Wolf, W, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 255–268.

from the inputs of an operator to its outputs transforms to: *i*) delaying by one iteration the consummation of the result produced by this operator, and *ii*) advancing by one iteration the consummation of the operands of that operator.

Applying retiming on a single loop leads to another code composed by three consecutive parts in the following order. The first one is a segment of code called *prologue*. The second one is a new loop which runs faster than the former one when the goal from retiming is optimizing timings. The third and last one is a segment of code called *epilogue*. The new loop can start executing once the execution of the *prologue* has terminated. The execution of the *epilogue* can begin once the new loop terminates executing.

There is more than one way to retime a loop for achieving a certain target. Depending on the retiming used, the code size of the *prologue* and *epilogue* can increase or decrease. Reducing the size of that code is very important in the case of embedded systems as well as of system-on-chip[1,3]. Both of these two kinds of systems have constraints on the memory size, and hence the code size must be reduced for them. The size of the code has also an implicit impact on both the execution time as well as the power consumption.The problem of reducing the code size has been widely addressed in the literature[2,4,5], but only few papers have recently addressed this problem when techniques like retiming has been used[1,3].

In the rest of this paper, we mean by "code size" the number of operations in the *prologue* and *epilogue* after retiming a loop. We address the problem of reducing the code size for retimed loops under timing and resource constraints. We formulate this problem mathematically and develop algorithms to optimally solve it. We provide a polynomial run-time algorithm to optimally solve the problem for the case of unlimited resources. In its general form, our target problem becomes NP-hard in the case of limited resources since to solve it, one needs to solve the problem of scheduling under timing and resource constraints which is already known as an NP-hard problem in its general form. We also propose an exact but not polynomial run-time algorithm to solve this latter version of our target problem.

There are many real-life examples of loop-intensive applications where the loops can be modeled as *for-type* loops. Moreover, the body of these loops does not contain conditional statements like *if-then-else*, etc. Such examples include digital filters like the correlator, the finite impulse response, and the infinite impulse response. In this paper, we focus on solving our target problem for these class of applications. This kind of applications is at the heart of many concrete embedded systems.

The rest of this paper is organized as follows. Section 2 shows how we model a loop. In Section 3, we give an introduction to basic retiming and

some of its related background required for our proposed approach. In Section 4, we define valid periodic schedules that we target in this paper and propose an algorithm to compute them. We develop in Section 5 an algorithm to generate the transformed code after applying a retiming on a single *for*-loop. In Section 6, we develop a polynomial-time exact method for computing a retiming leading to a small code size for the case of unlimited resources. In Section 7, we propose an exact but not polynomial-time algorithm to compute a retiming leading to a small code for the case of limited resources. Experimental results and conclusions are given in Sections 8 and 9, respectively.

2. CYCLIC GRAPH MODEL

In this paper, we are interested in *for-type* loops as the one in Fig. 1 (a). We assume that the body of the loop is constituted by a set of computational and/or assignment operations only (i.e., no conditional or branch instructions like for instance *if-then-else* is inside the body).

We model a loop by a directed cyclic graph $G = (V, E, d, w)$, where V is the set of operations in the loop's body, and E is the set of arcs that represent data dependencies. Each vertex $v \in V$ has a non-negative integer execution delay $d(v) \in N$, where N is the set of non-negative integers. Each arc $e_{u,v} \in E$, from vertex $u \in V$ to vertex $v \in V$, has a weight $w(e_{u,v}) \in N$, which means that the result produced by u at any iteration i is consumed by v at iteration $(i + w(e_{u,v}))$.

Fig. 1 presents a simple loop and its directed cyclic graph model. For Fig. 1 (b), the execution delay of each operation v_i is specified as a label on the left of each node of the graph, and the weight $w(e_{u,v})$ of each arc $e_{u,v} \in E$ is in bold. For instance, the execution delay of v_1 is 1 unit of time and the value 2 on the arc $e_{v1,v2}$ means that operation v_2 at any iteration i uses the result produced by operation v_1 at iteration $(i-2)$.

3. INTRODUCTION TO BASIC RETIMING

Let $G = (V, E, d, w)$ be a cyclic graph. Retiming[9] r is defined as a function $r : V \to Z$, which transforms G to a functionally equivalent cyclic graph $G_r = (V, E, d, w_r)$. The set Z represents natural integers.

The weight of each arc $e_{u,v}$ in G_r is defined as follows:

$$w_r(e_{u,v}) = w(e_{u,v}) + r(v) - r(u), \ \forall \ e_{u,v} \in E. \quad (1)$$

Since in the context of hardware the weight of each arc in G_r represents the number of registers on that arc, then we must have:

$$w_r(e_{u,v}) \geq 0 \ , \ \forall \ e_{u,v} \in E \ . \tag{2}$$

Any retiming r that satisfies inequality (2) is called a valid retiming. From expressions (1) and (2) one can deduce the following inequality:

$$r(u) - r(v) \leq w(e_{u,v}) \ , \ \forall \ e_{u,v} \in E \ . \tag{3}$$

Let us denote by $P(u,v)$ a path from node *u* to node *v* in *V*. Equation (1) implies that for every two nodes *u* and *v* in *V*, the change in the register count along any path $P(u,v)$ depends only on its two endpoints:

$$w_r(P(u,v)) = w(P(u,v)) + r(v) - r(u) \ , \ \forall \ u,v \in V \ , \tag{4}$$

where:
$$w(P(u,v)) = \sum_{ex,y \in P(u,v)} w(e_{x,y}) \ . \tag{5}$$

Let us denote by $d(P(u,v))$ the delay of a path $P(u,v)$ from node *u* to node *v*. $d(P(u,v))$ is the sum of the execution delays of all the computational elements that belong to $P(u,v)$.

A *0-weight path* is a path such that $w(P(u,v)) = 0$. The minimal clock period of a synchronous sequential digital design is the longest *0-weight path*. It is defined by the following equation:

$$\Pi = Max_{\forall u,v \in V}\{d(P(u,v)) \mid w(P(u,v)) = 0\} \ . \tag{6}$$

Two matrices called *W* and *D* are very important to the retiming algorithms. They are defined as follows[9]:

$$W(u,v) = Min\{w(P(u,v))\} \ , \ \forall u,v \in V \ , \tag{7}$$

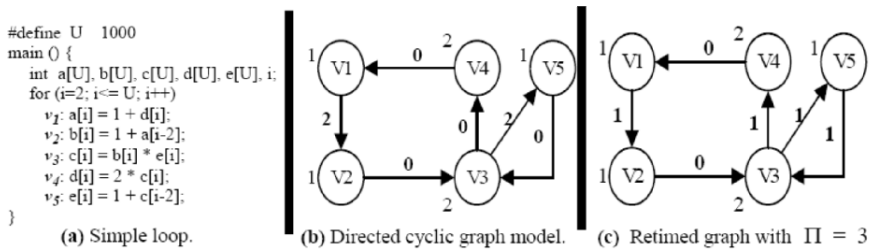
and
$$D(u,v) = Max\{d(P(u,v)) \mid w(P(u,v)) = W(u,v)\} \ , \ \forall u,v \in V \ . \tag{8}$$

The matrices *W* and *D* can be computed as explained in[9].

Minimizing the clock period of a synchronous sequential digital design is one of the original applications of retiming that are reported in[9]. For instance, for Fig. 1 (b), the clock period of that design is (see its definition above): $\Pi = 6$, which is equal to the sum of execution delays of computational elements $v_{i=1,...,4}$. However, we can obtain $\Pi = 3$ if we apply the following retiming vector $\{2,1,1,2,0\}$ to the vector of nodes $\{1,2,3,4,5\}$ in G. The retimed graph G_r is presented by Fig. 1 (c). Notice that the weight of each arc in G_r is computed using expression (1).

Once a retiming defined on Z is computed for solving a target problem, it can then be transformed to a non-negative retiming without impact on the solution of the problem. In the rest of this paper, we consider non-negative retiming only, unless it is explicitly specified otherwise. For the purpose of this paper, we extract from[9] the following theorem, which is also proved in[9].

Theorem 1: Let $G = (V, E, d, w)$ *be a synchronous digital design, and let* Π *be a positive real number. Then there is a retiming r of G such that the clock period of the resulting retimed design* G_r *is less than or equal to* Π *if and only if there exists an assignment of integer value r(v) to each node v in V such that the following conditions are satisfied: (1)* $r(u) - r(v) \leq w(e_{u,v}) \ \forall \ e_{u,v} \in E$ *, and (2)* $r(u) - r(v) \leq W(u,v) - 1 \ \ \forall u,v \in V \mid D(u,v) > \Pi$.



(a) Simple loop. **(b)** Directed cyclic graph model. **(c)** Retimed graph with $\Pi = 3$.

*Figure 1.*A simple loop, its directed cyclic graph model, and a retimed version of graph (b).

4. VALID PERIODIC SCHEDULE

Let $G = (V, E, d, w)$ be a directed cyclic graph modeling a loop. A schedule is a function $s : N \times V \to N$ that, for each iteration $k \in N$ of the loop, determines the start execution time $s_k(v)$ for each operation v of the loop's body. Here, N is the set of non-negative integers.

The schedule s is said to be *periodic* with period Π iff it satisfies (9):

$$s_k(v) = s_0(k) + k \cdot \Pi \,,\ \forall k \in N \,,\ \forall v \in V \,, \tag{9}$$

where $s_0(v)$ is the start execution time of the first instance of the operation v. Without loss of generality, we assume through this paper that:

$$s_0(v) \geq 1 \,,\ \forall v \in V \,. \tag{10}$$

In this paper, the schedule s is said to be *valid* iff it satisfies both *data dependency constraints* and *resource constraints*.

Data dependency constraints mean that a result computed by operation u can be used by operation v only after u has finished computing that result. In terms of start execution time, this is equivalent to the following inequality:

$$s_{(k\ +\ w(eu,\ v))}(v) \geq s_k(u) + d(u) \,,\ \forall k \in N \,,\ \forall e_{u,v} \in E \,. \tag{11}$$

Using equation (9), inequality (11) transforms to:

$$s_0(v) - s_0(u) \geq d(u) - \Pi \cdot w(e_{u,v}) \,,\ \forall e_{u,v} \in E \,. \tag{12}$$

In this paper, resource constraints mean that at any time, the number of operations that require execution on the processing unit number k must not exceed 1. For resource constraints, we handle in this paper each processing unit individually since this allows us to also realize the binding task (i.e., once the schedule is computed, we will automatically know on which resource each operation will execute).

The following notations and definitions will be used in this paper.

- $|F|$ The number of all the available functional units.
- $F(v)$ The set of labels of processing units that can execute v. It is:

$$F(v) = \{k \in \{1,...,|F|\} \mid v \text{ can execute on functional unit number } k\}.$$

- $x_{v,\,t(v),\,k}$ 0-1 unknown variable associated to each $v \in V$. This variable will be equal to 1 if operation v starts executing at time $t(v)$ on the functional unit number k, otherwise it will be equal to 0.

Notice that, in this paper, we assume that the functional units are not pipelined. Moreover, we are interested in the class of *valid periodic schedules* that satisfy the following constraint while Π must be the smallest possible.

$$1 \le s_0(v) \le \Pi + 1 - d(v), \ \forall v \in V. \tag{13}$$

If the operation v is executed using the functional unit number k, then the execution delay of v will be $d_k(v)$ instead of $d(v)$. Using $d_k(v)$ as an execution delay of v, the operation v will execute in the discrete interval of time $\{1,...,(\Pi + 1 - d_k(v))\}$. By using binary variables $x_{v,\,t(v),\,k}$ as well as other notations defined above and the expression (13), we can write each $s_0(v)$ as follows:

$$s_0(v) = \left(\sum_{k \in F(v)} \sum_{t(v)=1}^{\Pi+1-dk(v)} t(v) \cdot x_{v,\,t(v),\,k}\right), \ \forall v \in V, \tag{14}$$

where:

$$\sum_{k \in F(v)} \sum_{t(v)=1}^{\Pi+1-dk(v)} x_{v,\,t(v),\,k} = 1, \ \forall v \in V, \text{ and} \tag{15}$$

$$x_{v,\,t(v),\,k} \in \{0,1\}, \ \forall v \in V, \ \forall k \in F(v), \ t(v) = 1,...,(\Pi + 1 - d_k(v)). \tag{16}$$

In this paper, we are interested in identical binding for all the iterations of the loop. The operation v and all its future instances will execute on the same functional unit k. Hence, assuming the expressions (15) and (16) are available, the execution delay of v is :

$$d(v) = \left(\sum_{k \in F(v)} \sum_{t(v)=1}^{\Pi+1-dk(v)} d_k(v) \cdot x_{v,\,t(v),\,k}\right), \ \forall v \in V. \tag{17}$$

Notice that thanks to the binary variables $x_{v,\,t(v),\,k}$, the binding task will be implicitly carried out once the values of these variables become known.

Now, we will show how to derive a formal expression for the resource constraints. Any operation $v \in V$ which is executing at time t implies that v has started to execute somewhere in the discrete interval $\{Max(1,(t+1-d_k(v))),...,t\}$, which transforms to (18):

$$\sum_{k \in F(v)} \sum_{t(v)=Max(1,(t+1-dk(v)))}^{t} x_{v,\,t(v),\,k} = 1, \ \forall v \in V, \ t = 1,...,(\Pi + 1). \tag{18}$$

Using expression (13), any operation $v \in V$ must start executing no later than $(\Pi + 1 - d_k(v))$. Thus, equation (18) transforms to:

$$\sum_{k \in F(v)} \sum_{t(v)=Max(1,(t+1-dk(v)))}^{Min((\Pi+1-dk(v)),t)} x_{v,\,t(v),\,k} = 1, \ \forall v \in V, \ t = 1,...,(\Pi + 1). \tag{19}$$

The schedule s must be computed in such a way that at any time $t = 1,...,(\Pi + 1)$, the number of operations which are executing on any functional unit number k, $k = 1,2,...,|F|$, must not exceed 1 (recall that each functional unit is handled individually in this paper). This transforms to (20):

$$\sum_{\{\forall v \in V,\, k \in F(v)\}} \sum_{t(v)=Max(1,(t+1-dk(v)))}^{Min((\Pi+1-dk(v)),t)} x_{v,\,t(v),\,k} \le 1, \ k = 1,2,...,|F|, \ t = 1,...,(\Pi + 1). \tag{20}$$

Putting all these developments together, our target *valid periodic schedule* in this paper must satisfy the following set of constraints expressed

by (21)-to-(26). Notice that the constraint expressed by (13) is implicitly incorporated into (21)-to-(26) and hence it is implicitly satisfied.

- *Constraint#1: Each operation must start to execute at one and only one point of time.*

$$\sum_{k\in F(v)}\sum_{t(v)=1}^{\Pi+1-dk(v)} x_{v,\,t(v),\,k}=1\ ,\ \forall v\in V\ . \tag{21}$$

- *Constraint#2. Data dependencies must be satisfied.*

$$\left(\sum_{k\in F(v)}\sum_{t(v)=1}^{\Pi+1-dk(v)} t(v)\cdot x_{v,\,t(v),\,k}\right)-\left(\sum_{k\in F(u)}\sum_{t(u)=1}^{\Pi+1-dk(u)} t(u)\cdot x_{u,\,t(u),\,k}\right)\geq \left(\sum_{k\in F(u)}\sum_{t(u)=1}^{\Pi+1-dk(u)} d_k(u)\cdot x_{u,\,t(u),\,k}\right)-\Pi\cdot w(e_{u,\,v})\ ,\ \forall e_{u,\,v}\in E\ . \tag{22}$$

- *Constraint#3. Only one operation can execute on a functional unit at any point of time.*

$$\sum_{\{\forall v\in V,\,k\in F(v)\}}\sum_{t(v)=Max(1,(t+1-dk(v)))}^{Min((\Pi+1-dk(v)),t)} x_{v,\,t(v),\,k}\leq 1\ ,\ k=1,2,...,|\,F\,|\ ,\ t=1,...,(\Pi+1). \tag{23}$$

- *Constraint#4. It is just a re-writing of the expression (16). So we have:*

$$x_{v,\,t(v),\,k}\geq 0\ ,\ \forall v\in V\ ,\ \forall k\in F(v)\ ,\ t(v)=1,...,(\Pi+1-d_k(v)). \tag{24}$$

$$x_{v,\,t(v),\,k}\leq 1\ ,\ \forall v\in V\ ,\ \forall k\in F(v)\ ,\ t(v)=1,...,(\Pi+1-d_k(v)). \tag{25}$$

$$x_{v,\,t(v),\,k}\in N\ ,\ \forall v\in V\ ,\ \forall k\in F(v)\ ,\ t(v)=1,...,(\Pi+1-d_k(v)). \tag{26}$$

<u>*MinPeriod Algorithm*</u>: To compute a valid periodic schedule that satisfies the constraints expressed by (21)-to-(26) (which implicitly include (13) as well) while minimizing Π, then one can initialize Π to a lower bound (0 by default) and iteratively do $\Pi=\Pi+1$ (or do a binary search) until the system expressed by (21)-to-(26) can be solved. Π must be fixed before solving this system to allow for the linearity of (21)-to-(26). The tool [10] can be used to solve this system at each iteration, as we did for the algorithm in Section 7.

5. CODE GENERATION AFTER RETIMING

In the rest of this paper, to avoid repetition, we denote by Π the length of the longest *0-weight path* in a given loop (including the retimed loop). Assuming no resource constraints at this time, Π can be minimized by using one of the retiming algorithms for clock period minimization proposed in[9].

Our objective in this section is to show how to generate the transformed code after applying a retiming on a given loop to minimize Π. The transformed code is constituted by three parts: *prologue*, *new loop*, and *epilogue*. Recall that the *prologue* is the segment of the original code that must be executed before a repetitive processing appears that corresponds to the *new loop*. The *epilogue* is the segment of the original code that cannot be executed by the *new loop* and the *prologue*. Let us use the example of a loop in Fig. 1(a). As provided in Section 3, the value of Π for this loop is 6 units

of time. By applying the retiming vector $\{2,1,1,2,0\}$ on Fig. 1 (b), we can get the retimed graph given by Fig. 1 (c), where the value of Π is now 3 units of time. The transformed code after applying this retiming is given in Fig. 2(a).

Let $M = Max\{r(v), \forall v \in V\}$. With the help of Fig. 2, one can easily double-check that the **P**rologue, the new **L**oop and the **E**pilogue for the retimed loop can be obtained by the algorithm **PLE** below.

Algorithm: PLE
/* The index i of the original loop is in [L, U]. $M = Max\{r(v), \forall v \in V\}$ */
1- Prologue Generation
1.1 $i = L$.
1.2 While ($i < (L + M)$) do
$\forall v \in V$, if ($(i + r(v)) < (L + M)$) then generate a copy of v at iteration i as it was in the original loop's body. And, do: $i = i + 1$.
2- New Loop Generation
2.1 The index of the new loop is in [(L + M), U].
2.2 Determine the body of the new loop as follows. $\forall v \in V$, generate a copy of v where the index of each array in v of the original loop's body has now to be decreased by $r(v)$.
3- Epilogue Generation
3.1 $i = U + 1$.
3.2 While ($i \leq (U + M)$)
$\forall v \in V$, if ($(i - r(v) \leq U)$) then generate a copy of v by evaluating its expression derived from Step 2.2. And, do: $i = i + 1$.
End of the algorithm PLE.

6. COMPUTING A RETIMING WITH REDUCED CODE SIZE UNDER TIMING CONSTRAINTS FOR THE CASE OF UNLIMITED RESOURCES

Recall that $\forall v \in V$, $r(v)$ is the value assigned by the retiming r to the vertex v. Also, notice that $M = Max\{r(v), \forall v \in V\}$. With the help of the *PLE* algorithm above and Fig. 2(a) and 2(b) below, one can deduce that for the code after retiming a loop, we have:

- The *prologue* contains $(M - r(v))$ copies of the operation modeled by $v \in V$.
- The *epilogue* contains $r(v)$ copies of the operation modeled by $v \in V$.
- For the code composed by the *prologue* and the *epilogue*, the number of instances of each operation is exactly M times.
- If the original loop executes K times, the new loop executes $(K - M)$ times.

Consequently, to reduce the code size for the retimed loop, one needs to reduce the value of M. The value of M depends on the retiming used and is not unique. Indeed, we showed in the previous sections that the value of Π for the loop in Fig. 1 can be reduced from 6 units of time to 3 units of time by applying the retiming vector $\{2,1,1,2,0\}$ on Fig. 1 (b). But, the retiming vector $\{2 + x, 1 + x, 1 + x, 2 + x, 0 + x\}$ can also be applied to obtain

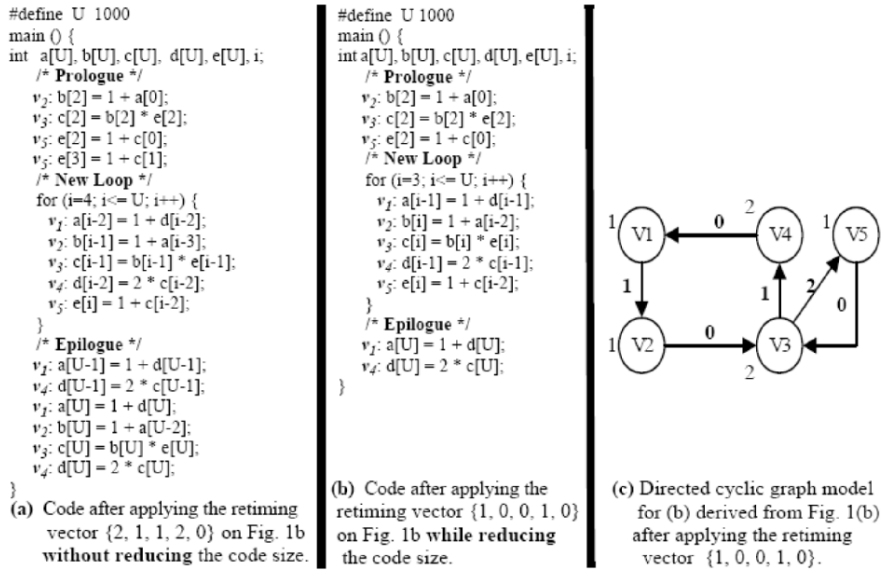


Figure 2. Transformed code after retiming a loop.

$\Pi = 3$ units of time, where x is a non-negative integer. For the first retiming vector, we have $M = 2$, and for the second one we have $M = 2 + x$. The last retiming vector is then not a good choice and can lead to completely unroll the loop if $x = U - 2$. While the first retiming vector (i.e., $\{2,1,1,2,0\}$) is better than the second one (i.e., $\{2 + x,1 + x,1 + x,2 + x,0 + x\}$) for this example, it is not a good one as well. Indeed, one can achieve $\Pi = 3$ units of time by applying the retiming vector $\{1,0,0,1,0\}$ on Fig. 1 (b). In this case, the resulting retimed graph is given by Fig. 2(c), and the transformed code is provided on Fig. 2(b). For this vector, we have $M = 1$. Also, by comparing Fig. 2(a) and Fig. 2(b), one can easily notice that the size of the code has been reduced (which is consistent with the statements in the previous paragraph).

Recall that we are assuming non-negative retiming as explained in the end of Section 3. To determine a retiming that leads to a small value of *M*, one can add a variable upper bound β on the retiming value for each $v \in V$ as in (27).

$$r(v) \le \beta, \ \forall v \in V. \tag{27}$$

Using this upper bound β, we propose the following method to compute a retiming that leads to a small value of *M* (and thus to a code with a small size).

We can extend the system of inequalities in Theorem 1 by first adding an upper bound on the value of the retiming function as we did in inequality (27), but assuming without loss of generality that retiming will take non-negative values. And, second minimizing the value of the unknown variable

β. This leads to the Integer Linear Program (ILP) composed by (28)-to-(32) below.

In the ILP (28)-to-(32), recall that Π denotes the length of the longest *0-weight path* in a given loop (including the retimed loop). Also, notice that in the context of clocked sequential circuits, Π is the clock period. Hence, to solve this ILP, we must first compute the value of Π if it is not provided by the user. To do so, a retiming using one of the algorithms in[9] (for clock period minimization) can be first carried out.

$$Minimize(\beta) \tag{28}$$

Subject to:

$$r(u) - r(v) \le w(e_{u,v}),\ \forall e_{u,v} \in E \tag{29}$$

$$r(u) - r(v) \le W(u,v) - 1,\ \forall u,v \in V \mid D(u,v) > \Pi \tag{30}$$

$$r(v) \le \beta,\ \forall v \in V \tag{31}$$

$$\beta, r(v) \in N,\ \forall v \in V \tag{32}$$

The ILP (28)-(32) can be solved using the linear programming solver in[10].

Lemma. The ILP (28)-(32) can be optimally solved in polynomial run-time.

Proof: Omitted due to the space limitation.

7. COMPUTING A RETIMING WITH REDUCED CODE SIZE UNDER TIMING AND RESOURCES CONSTRAINTS

In this case, one needs to also perform a scheduling under timing and resource constraints. Since we are dealing with loops, this schedule is periodic. Hence, one needs to compute a valid periodic schedule while minimizing timings as we have presented in Section 4. For instance, to compute this schedule while minimizing its period, one can use the *MinPeriod* algorithm outlined at the end of Section 4. However, notice that this algorithm will produce a *minimal value* for the period Π *relative to the input graph* G. This algorithm might miss the *absolute minimal value* of Π since it does not alter the weight of arcs in the graph (in fact, any algorithm that does not alter the weight of arcs in the graph might miss the *absolute minimal value* of Π). Altering the weight of arcs in the graph is the task of retiming. For instance, assume we have three resources: two identical adders and one multiplier. Suppose that the execution delays of the adder and multiplier are 1 and 2 units of times, respectively. So for these resources, applying the *MinPeriod* algorithm on the graph in Fig. 1(b) will lead to a minimal value of $\Pi = 6$ units of time. However, we can get $\Pi = 3$ units of time if we apply the *MinPeriod* algorithm on either the retimed graph in Fig.

1(c) or the retimed graph in Fig. 2(c). Recall that the retiming used for producing the retimed graph in Fig. 2(c) is better than the retiming used to produce the retimed graph in Fig. 1(c), since it allows to reduce the code size as it was explained in Section 6.

To get an *absolute minimal value* of Π, one needs to *optimally unify* scheduling and retiming. To achieve this optimal unification, we propose to extend the *MinPeriod* algorithm. The extended version (see the *MinPeriod-WithReducedCodeSize* below) of this algorithm should operate on a *dynamically retimed graph* $G_r = (V, E, d, w_r)$. The graph G_r is as the one defined in Section 3 except that in equation (1), the values $r(u)$ and $r(v)$ will be calculated on the fly during the schedule determination; this is why we said: *dynamically retimed graph*. Of course the dynamically retimed G_r graph must be *functionally equivalent* to the original graph G, which means that the inequality (3) must hold. Moreover, the inequality (27) must also be respected since it will allow to produce a retiming with a reduced code size. Since we are interested in non-negative retiming, then the following expression (33) must be considered as well.

$$r(v) \in N, \ \forall v \in V. \tag{33}$$

And, for the *MinPeriodWithReducedCodeSize* algorithm, data dependencies expressed by (22) now transform to the expression (34) below, since the weight of each arc $e_{u,v} \in E$ is no longer $w(e_{u,v})$ but it is now equal to $\left(w(e_{u,v}) + r(v) - r(u)\right)$. Moreover, we have two parameters to minimize: Π and β. So, we need to first minimize Π for a large value of β in order to get an absolute minimal value of . Next, we need to minimize β to achieve this latter value of Π.

$$\left(\sum_{k\in F(v)}\sum_{t(v)=1}^{\Pi+1-dk(v)} t(v)\cdot x_{v,t(v),k}\right) - \left(\sum_{k\in F(u)}\sum_{t(u)=1}^{\Pi+1-dk(u)} t(u)\cdot x_{u,t(u),k}\right) \geq$$
$$\left(\sum_{k\in F(u)}\sum_{t(u)=1}^{\Pi+1-dk(u)} d_k(u)\cdot x_{u,t(u),k}\right) - \Pi\cdot\left(w(e_{u,v}) + r(v) - r(u)\right), \ \forall e_{u,v} \in E. \tag{34}$$

Algorithm: *MinPeriodWithReducedCodeSize*

Inputs: Cyclic graph $G = (V, E, d, w)$, the set of available functional units, and the set of processing units that can execute each operation $v \in V$.

Outputs: Schedule, binding, and minimal values of Π and β which is the maximal value of retiming.

Begin

1. The optimal value of Π is in the interval [*L*, *U*]. *L* is the delay of the fastest functional unit. *U* is *the number of nodes* in the graph G *times the execution delay* of the slowest functional unit. If a lower bound on the minimal value of Π is not provided by the user, then let: $\Pi = L$.
2. Using a linear programming solver like the one in[10], solve the system expressed by: (3), (21), (23)-to-(26) and (33)-(34).
3. If the system expressed by(3), (21), (23)-to-(26) and (33)-(34) has a solution, then go to Step 4.
   Else $\Pi = \Pi + 1$ (or do a binary search in [*L*, *U*] to determine Π). And go to Step 2.
4. Let $L = 0$, and $U = K$. Let Π be fixed to the value found in Step 3.
5. Let $\beta = \lceil (U + L)/2 \rceil$. If β cannot be reduced further then go to Step 8.
6. Using a linear programming solver like the one in[10], solve the system expressed by: (3), (21), (27), (23)-to-(26) and (33)-(34).
7. If the system expressed by (3), (21), (27), (23)-to-(26) and (33)-(34) has a feasible solution, then

7.1. From the solution found in Step 6, extract the values for the schedule and the binding as well as the values of Π and β, and save them.

7.2. Let $U = \beta$. And go to Step 5.

else Let $L = \beta$. And go to Step 5.

8. Report the result extracted in Step 7.1 and Stop.

End of the algorithm *MinPeriodWithReducedCodeSize.*

8. EXPERIMENTAL RESULTS

Although the proposed approaches in this paper solve optimally the target problem, we however found it logic to experimentally test theirs effectiveness in terms of: *i*) the impact of retiming on reducing the length of the critical paths (*i.e., the length of the longest 0-weight path denoted as* Π *in this paper*) in the case of unlimited resources, *ii*) the impact of retiming on reducing the period's length of the schedule for the case of limited resources, and *iii*) the impact of controlling the values of retiming on reducing the code size of the retimed loops. To this end, some real-life digital filters are used as input loops to be optimized. The body of these loops (filters) is composed by additions and multiplications operations only. The names of these filters and other designs are labeled on Fig. 3 and 4 as follows. L1: The exemple given on Fig. 1b. L2: Correlator. L3: FIR Filter. L4: Biquadratic Filter. L5: Polynomial Divider. L6: Three Tape Non-Recursive Digital Filter. L7: Lowpass Reverberator First Order with Feedback. L8: Allpass Reverberator Canonical Form. L9: Coupled form Sine-Cosine Generator. L10: Digital Cosinusoidal Generator. L11: Resonator Filter.

For the results on Fig. 3 and 4, we developed a C++ tool which, from an input graph modeling the target loop, automatically generates the expressions of the ILP (28)-to-(32). In this tool, we also implemented the proposed algorithm *MinPeriodWithReducedCodeSize*. This tool also automatically generates the expressions of the system to be solved in the steps 3 and 7 of the *MinPeriodWithReducedCodeSize* algorithm, and needs to be re-run until the *MinPeriodWithReducedCodeSize* finds the optimal values to be values to be computed.

For results in Fig. 3, we assume unlimited resources. We first compute the value of Π without any retiming. Next, we apply a retiming for minimizing Π, by implementing and running an algorithm for clock period minimization from[9]. This latter step is done twice: *i*) we apply a retiming for minimizing Π without controlling the largest value of the retiming, and *ii*) we apply a retiming to achieve the same minimal value of Π but we minimize the largest value of the retiming (so, in this case we solve the ILP (28)-to-(32)). As it can be noticed from Fig. 3, it was possible to reduce the value of Π for some input designs. For the set of input designs in this

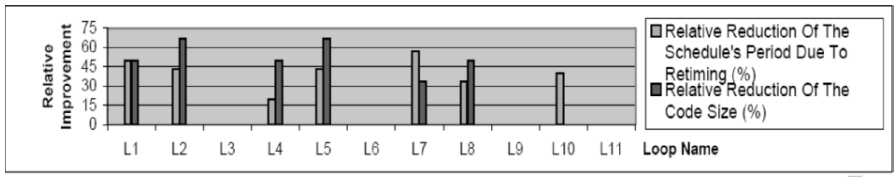


Figure 3. Case of unlimited resources.

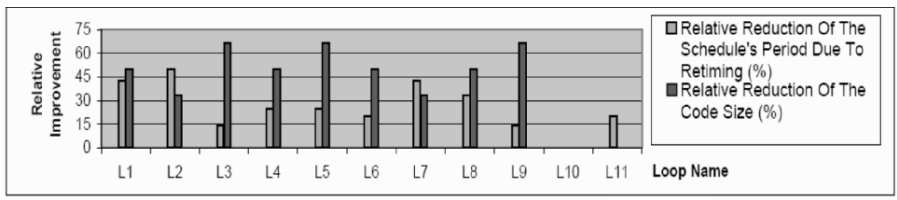


Figure 4. Case of limited resources.

experimentation, the maximal relative reduction of Π is 57%. When it was possible to reduce Π by applying a retiming, then by controlling the values of retiming, we were able to reduce the code size of the retimed loops by as high as 66.67%.

For results in Fig. 4, we assume limited resources. In this case, we suppose that we have an hypothetical processor with 2 non-identical adders and 2 non-identical multipliers. For our adders, one of them has execution delay equal to 1 unit of time and the second one has execution delay equal to 2 units of time. For our multipliers, one of them has execution delay equal to 2 units of time and the second one has execution delay equal to 3 units of time. Next, for this hypothetical processor, we first compute a valid periodic schedule with a minimal period Π but without any retiming. For doing this, the C++ tool executes the algorithm *MinPeriod*. Then, we compute a valid periodic schedule with a minimal period Π but in this case we perform retiming as well. For this latter case, the C++ tool executes the algorithm *MinPeriodWithReducedCodeSize* and we proceed in two steps: *i*) we apply a retiming without controlling its largest value (in this case, β has no effect in the algorithm *MinPeriodWithReducedCodeSize*), and *ii*) we apply a retiming while minimizing its largest value (by minimizing β in this case). As it can be noticed from Fig. 4, except for the design L10, it was possible to reduce the value of Π for all the other input designs. For the set of input designs in this experimentation, the relative reduction of Π ranges from 14% to 50%. When it was possible to reduce Π by applying a retiming, then by controlling the values of retiming, we were able to reduce the code size of the retimed loops by as high as 66.67%. The run-time of the algorithms *MinPeriod* and *MinPeriodWithReducedCodeSize* is dominated by the run-time of solving the systems of constraints they contain. The run-time for solving these systems of constraints was less than 4 seconds for these experimental results. The bounds we have introduced in the expressions of

these constraints (i.e., see the *Min* and *Max* in some expressions) are helped in achieving such small run-times.

9. CONCLUSIONS

Retiming has been originally proposed as an optimization technique for clocked digital circuits. It has been successfully applied for optimizing loop-intensive programs. Decreasing the code size for retimed loops is of great importance in particular for memory-constrained system-on-chip and embedded systems. The size of the code has also an implicit impact on both the run-time and the power consumption. This paper addressed the problem of reducing the code size for retimed software loops under timing and resource constraints. We mathematically formulated this problem and devised exact algorithms to optimally solve it. Experimental results have re-confirmed the importance of solving this problem. For unlimited resources, the exact polynomial-time method can be used for source-to-source code optimization. For limited resources, the approach can be used to design optimized libraries, to develop heuristics and to test their effectiveness.

REFERENCES

1. Zhuge Q., Xiao B., and Sha, E.H.M.: Code size reduction technique and implementation for software-pipelined DSP applications. ACM Trans. on Embedded Comput. Syst., Vol.2, No.4, pp. 590-613 (2003).
2. Benini L., Macii A., Macii E., and Poncino M.: Minimizing memory access energy in embedded systems by selective instruction compression. IEEE Transactions on Very Large Scale Integration Systems, Vol.10, No. 5, pp 521-531 (2002).
3. Granston E., Scales R., Stotzer E., Ward A., and Zbiciak J.: Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. Proceedings 3rd IEEE/ACM Workshop on Media and Streaming Processors, pp. 29-38 (2001).
4. Sung W., and Ha S.: Memory efficient software synthesis with mixed coding style from dataflow graphs. IEEE Trans. on Very Large Scale Integration Systems, Vol. 8, No. 5, pp 522-526 (2000).
5. Lekatsas H, Henkel J., and Wolf W.: Code compression for low power embedded systems design. Proc. of Design Automation Conference, pp. 294–299 (2000).
6. Nemhauser G.L., and Wolsey L.A.: Integer and Combinatorial Optimization, John Wiley and Sons Inc., ISBN 0-471-35943-2 (1999).
7. Fraboulet A., Mignotte A., and Huard G.: Loop alignment for memory accesses optimization. Proc. of 12th International Symposium on System Synthesis, pp.71-77 (1999).
8. Chao L.F., LaPaugh A.S., and Sha E.H.M.: Rotation scheduling, A loop pipelining algorithm. IEEE Trans. on Computer-Aided Design of Integrated Circ. & Syst., Vol.16, No.3, pp. 229-239 (1997).
9. Leiserson C.E., and Saxe J.B.: Retiming Synchronous Circuitry. Algorithmica, pp. 5-35 (1991).
10. The LP_Solve Tool: ftp://ftp.ics.ele.tue.nl/pub/lp_solve/

IDENTIFICATION AND REMOVAL OF PROGRAM SLICE CRITERIA FOR CODE SIZE REDUCTION IN EMBEDDED SYSTEMS

Mark Panahi, Trevor Harmon, Juan A. Colmenares*, Shruti Gorappa, and Raymond Klefstad
Department of Electrical Engineering and Computer Science
University of California, Irvine, CA 92697, USA
{mpanahi, tharmon, jcolmena, sgorappa, klefstad}@uci.edu

Abstract: Code shrinking is a technique for reducing the size of a software system by detecting and eliminating unused features. It is a static approach that is limited to information known at build time. We demonstrate how to extend code shrinking to take dynamic behavior into account. Our technique, Slice Criteria Identification and REmoval (SCIRE), combines conditional compilation and code shrinking techniques with information provided by the developer about run-time behavior. This integration allows SCIRE to achieve code reduction that is close to optimal while remaining mostly automatic. Our results show that SCIRE reduces code size substantially: In many cases, removal of unused features shrinks middleware footprint to less than 5% of its original size, allowing CORBA and other large software libraries to be used in embedded systems.

1. STATIC FOOTPRINT MINIMIZATION TECHNIQUES

Over the years, three basic techniques have emerged for reducing the size of executable code ("footprint"):

1. **Specification subsetting**: Removal of infrequently used features from a standard shared library, API, or class hierarchy.
2. **Conditional compilation**: Manual removal of features on a per-application basis using preprocessor directives.

*Also with the Applied Computing Institute, Faculty of Engineering, University of Zulia.

Panahi, M., Harmon, T., Colmenares, J.A., Gorappa, S., Klefstad, R, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 269–278.

3. **Code shrinking**: Automatic removal of dead code by pruning a graph of all possible execution paths.

Each approach offers a unique set of strengths and weaknesses, and therefore any innovation beyond these existing techniques requires a careful comparison to discover avenues of improvement. Toward that end, we have identified three criteria for evaluating each approach:

- **Flexibility** *(Criterion 1).* A flexible approach gives the developer freedom to choose the features necessary to accommodate design requirements.
- **Maintainability** *(Criterion 2).* A maintainable approach is one that remains manageable as the size of the application grows. Meeting this criterion requires some form of automation to avoid loss of productivity as the code base increases.
- **Run-time-awareness** *(Criterion 3).* A "run-time-aware" approach takes into account knowledge of application-specific requirements and explicitly removes features that will (according to the developer) never be required at run-time.

For the remainder of this section, we examine the three approaches to footprint reduction from an application developer's standpoint and discuss if and how they meet the three criteria listed above.

Specification Subsetting. With this approach, software architects specify reduced profiles (e.g., an API or class hierarchy) of existing software libraries with an eye toward resource-constrained embedded devices. These reductions are obtained by removing features from standard library specifications. This task necessarily implies a trade-off between the conflicting goals of small footprint and usability. Thus, the memory footprint reduction obtained from specification subsetting varies widely and depends heavily on the original profiles and the objectives established by the specification.

A well-known example is the Java 2 Platform, Micro Edition (J2ME). By removing large portions of the standard Java class library and providing a simplified virtual machine interpreter, J2ME meets the stringent requirements of many embedded systems. However, J2ME's main disadvantage is that the reduced profiles are defined beforehand, preventing adaptation to unanticipated application requirements *(Criterion 1).*

Moreover, specification subsetting does not take advantage of the knowledge of a system's run-time conditions *(Criterion 3).* In practice, features in reduced profiles are considered fixed, and even if an application developer can guarantee that certain features will never be used during execution, such features cannot be removed.

Conditional Compilation. Conditional compilation involves a preprocessor examining compile-time configuration flags to decide which portions of a shared library belong in the final executable package. It is most often applied in C/C++ using the `#ifdef` directive to allow retargeting of libraries to different platforms.

Because conditional compilation is applied on a per-application basis, developers can include only the features they need. It imposes no restrictions on accommodating application requirements *(Criterion 1)* and allows the developer to exploit knowledge of the system's run-time conditions *(Criterion 3)*. Thus, in theory, conditional compilation can reduce the footprint of a program to the absolute minimum. In practice, however, obtaining the theoretical minimum code size requires enormous effort, and therefore developers routinely accept sub-optimal results when using this technique.

Code Shrinking. Many compilers include optimization algorithms to reduce the footprint of a program. These algorithms typically eliminate redundant and unused code based on an *intra*-procedural analysis. However, they can easily fail to identify unused code due to a lack of information necessary for a full *inter*-procedural analysis. When parsing the shared library of an API, for example, compilers have no way of knowing which of the API's functions will (or will not) be called by a given application.

For inter-procedural elimination, there exist footprint optimization tools that perform whole-program analysis. These optimizers, also known as *code shrinkers*, reduce footprint by analyzing an entire program, including any shared libraries that it uses, and removing unnecessary ("dead") portions of code. Code shrinkers are able to detect this dead code by building a call graph of all possible execution paths of a given program. Any classes and methods not in the graph are removed, as shown in Figures 1 and 2. This memory reduction process is called *code shrinking* or *dead code elimination*.

```
public class Hello {
  public Hello(boolean b) {
    if (b) methodA();
    else methodB();
  }
  private void methodA() {
    System.out.println("Method A");
  }
  private void methodB() {
    System.out.println("Method B");
  }
  private void methodC() {
    System.out.println("Method C");
  }
  public static void main(String[] args) {
    Hello h = new Hello(args.length > 0);
  }
}
```

Figure 1. Java compilers generate code for method C, but code shrinkers correctly identify it as dead code and remove it.

A code shrinker's ability to remove unused features from a program is effective but still sub-optimal. Because it has no knowledge of run-time conditions, features may appear to the shrinker as having structural

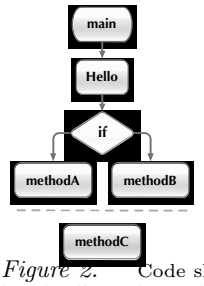


Figure 2. Code shrinkers build call graphs, such as this graph of Figure 1, to conclude that method C is never called and can be removed.

Technique	Flexible *(Crit. 1)*	Maintainable *(Crit. 2)*	Run-time-aware *(Crit. 3)*
Specification subsetting		✓	
Conditional compilation	✓		✓
Code shrinking	✓	✓	

Figure 3. A comparison of footprint reduction techniques.

dependencies even though they are independent according to the application's true run-time behavior. In other words, code shrinkers will not remove any code that could *potentially* be executed at run-time *(Criterion 3)*.

Qualitative Comparison. Table 3 summarizes the three footprint reduction techniques according to our criteria. Note that none of these techniques is able to meet all criteria. We observe, however, that if the code shrinking approach could be made run-time aware, it would meet all three criteria. This observation is the basis for our SCIRE approach, as described in the following section.

2. SCIRE

SCIRE, *Slice Criteria Identification and REmoval*, is our technique for enhancing the ability of code shrinkers to discover dead code by incorporating knowledge of run-time behavior. We first explain SCIRE's foundation in program slicing [Weiser, 1981] and then how it may be applied to the problem of footprint reduction.

Program Slicing. Program slicing is a well-known software analysis technique that is used to identify code that influences or is influenced by a point of interest in a program. The point of interest is called the *slice criterion* and could be a variable or any program statement. The code that is affected by the criterion is known as the *forward slice*, and the code that affects the criterion is known as the *backward slice*.

Code shrinking tools mainly use the forward slicing technique to remove unused code. However, code shrinking tools are based only on *static program slicing*, where the slice is determined based on the static program structure and does not include any run-time information. In

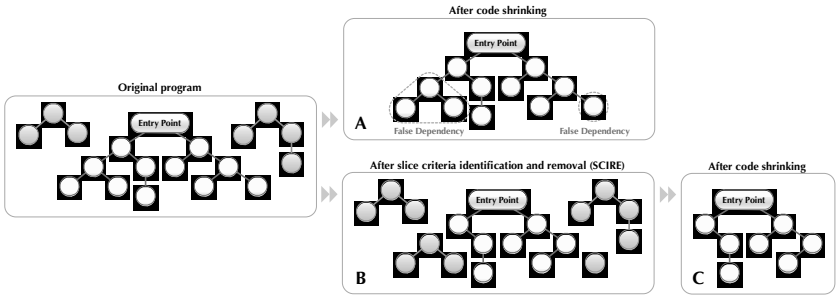


Figure 4. Removing syntactic dependencies (that is, the slice criteria) between features can enhance the effect of code shrinking.

contrast, *dynamic program slicing* [Agrawal and Horgan, 1990] leverages run-time information—in addition to static structure—to identify the true program slice for a given execution. Dynamic slices are therefore smaller in most cases and no larger than their corresponding static slices. Their precision varies depending on the slicing approach [Agrawal and Horgan, 1990, Zhang et al., 2003]. In our approach, we rely on the developer's run-time knowledge of a given slice criterion to apply dynamic slicing. As a result, only the code truly necessary for program execution is retained in the final footprint.

Application of Program Slicing to Footprint Reduction. Program slicing techniques have traditionally been used to examine and understand program execution, for debugging, and for testing. In this paper, we present a new application of this technique to achieve code footprint reduction. To the best of our knowledge, dynamic program slicing has not previously been used to perform code reduction.

Our approach, Slice Criteria Identification and REmoval (SCIRE), is simple in that it only requires the identification of program slice criteria, rather than the entire program slice associated with a feature, as with a purely conditional compilation (PCC) approach. Through the identification and optional removal of such dependencies, SCIRE enhances the code shrinking approach. The key observation is that certain slice criteria, such as an `if` statement, may mark the start of a program slice such that the decision to execute these slices cannot be made until run-time.

Figure 4 presents a generalized diagram of this process. Each circle, or node, represents a basic program block. The links between nodes represent dependencies between the blocks, such as `if` statements or function calls. Analysis begins at the node marked *Entry Point*, usually the standard `main` function. The shaded nodes are program blocks that

have no dependency on the entry point and can thus be removed through code shrinking, as shown in part A. The nodes labeled *False Dependency* are reachable from the entry point, but they are not actually needed by this application at run-time.

In our alternative process, beginning in part B, SCIRE is applied to remove the slice criteria binding the false dependencies to the entry point. When code shrinking is subsequently applied (part C), it is able to remove both the truly independent program slices and the falsely dependent program slices. As more slice criteria are identified and removed, the cumulative effect on the overall footprint can be significant, typically around 30-50% more than code shrinking alone, according to our experimental results.

3. IDENTIFYING PROGRAM SLICE CRITERIA

The effectiveness of SCIRE depends on our ability to identify slice criteria that bind unnecessary features in software libraries to a given application. Finding many of the features in libraries, such as middleware and foundation classes, is difficult because code associated with these features is typically scattered throughout several locations. In addition, the codebase may be very large and complex, making manual identification of slice criteria extremely difficult and time-consuming.

To overcome these drawbacks, we have developed our own tool called Shiv for SCIRE that is based not on aspects but on static program slicing. This change allows Shiv to focus on SCIRE's overall goal of footprint reduction. By taking into account the code size of each program slice, it can provide a visual representation of slices along with their corresponding contribution to the overall footprint. The developer can then use this visualization to more easily identify large program slices as candidates for removal.

Shiv offers the developer two choices for visualizing the call graph: a tree or a treemap. The tree is a traditional node-link visualization, where the root of the tree is the chosen slice criterion, each node is a method in the program, and each link is a dependency from one method to another (i.e., a method call). The treemap alternative represents the same tree structure, but it does so using a space-filling algorithm [Johnson and Shneiderman, 1991] that takes into account the code size of each method in the call graph. This allows the largest program slices to be identified easily.

We illustrate the benefits of Shiv using the canonical bubblesort algorithm as an example. When Shiv is provided the program in Fig-

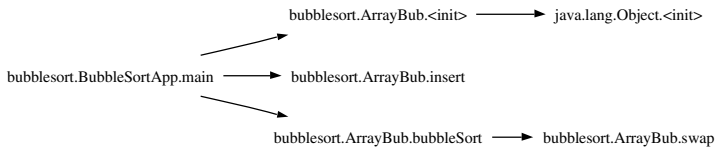


Figure 6. A node-link tree visualization generated by Shiv of the program in Figure 5.

ure 5 as input, it generates the tree shown in Figure 6. These figures show the relationship between a program's source code and its visualization in Shiv. For example, the `main` method calls `ArrayBub.insert`, `ArrayBub.bubbleSort`, and `ArrayBub`'s constructor, and therefore each of these calls appears as a child of `main`.

```
class ArrayBub {
  private long[] a; private int nElems;
  public ArrayBub(int max) {
    a = new long[max];
    nElems = 0;
  }
  public void insert(long value) {
    a[nElems] = value;
    nElems++;
  }
  public void bubbleSort() {
    for (int out = nElems-1; out > 1; out--)
      for (int in = 0; in < out; in++)
        if ( a[in] > a[in+1] )
          swap(in, in+1);
  }
  private void swap(int one, int two) {
    long temp = a[one];
    a[one] = a[two];
    a[two] = temp;
  }
}
class BubbleSortApp {
  public static void main(String[] args) {
    ArrayBub arr = new ArrayBub(100);
    arr.insert(77); arr.insert(99); arr.insert(44);
    arr.bubbleSort();
  }
}
```

Figure 5. This simple bubblesort program demonstrates the effectiveness of our slice criteria identification tool, Shiv. Visualizations of this program can be seen in Figures 6, 7, and 8.

The true importance of Shiv can be seen if a slight change is made to the Figure 5 code. If a single `println` statement is added, the call graph explodes into the tree shown in Figure 7. The Shiv output reveals that the single slice criterion of the `println` statement is the source of an extremely large program slice. (In this case, the large size is due to the security checks that Java requires for all program output.) We can also see possible features starting to emerge, exhibited by clusters in the tree. These clusters indicate program slices that could be modularized or perhaps removed entirely, depending on the true run-time behavior of the program using the slice.

The tree in Figure 7 is informative, but recall that the motivation of Shiv, and our SCIRE technique in general, is footprint reduction. For such a goal, a simple node-link diagram does not capture information about code size. For example, a collection of very small

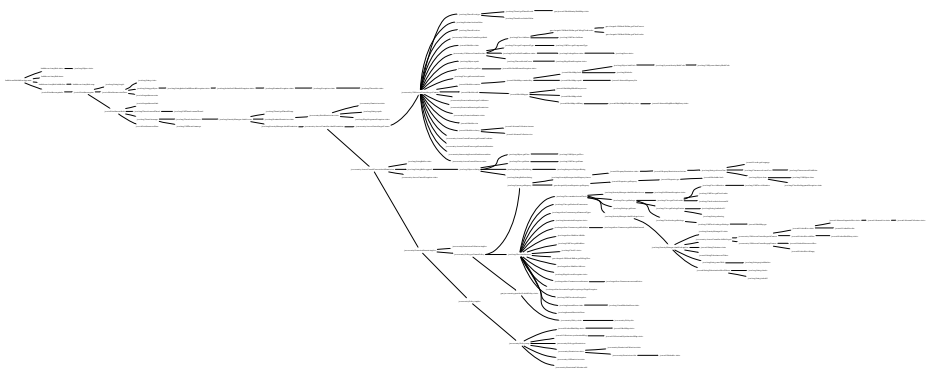


Figure 7. A node-link tree visualization generated by Shiv after adding a `println` statement to the program in Figure 5. Although the labels are not legible here, the structure shows that, in comparison with Figure 6, the size of the call graph has exploded with the addition of a single line of code.

methods may appear in the tree as a large cluster, while a single large method that takes up even more space would appear only as a single node. This potential to mislead the developer compelled us to offer an alternative visualization in Shiv: the treemap.

Using the Treemap 4.0 program [Baehrecke et al., 2004], Shiv produces a visualization of the program slice that quantifies the largest methods contributing to that slice. For example, Figure 8 shows the same bubble-sort program of Figure 5, again with an added `println` statement. By examining the branch labels of this squarified treemap, we can see that the largest contributors to code size include the `RuntimePersmission` constructor and the `HashMap.put` method, for they are represented by the largest blocks. Looking further up the tree, we note that the `checkAccess` method corresponds to an extremely large block. Thus, security checks make up the vast majority of the static footprint of this program.

Based on this information, the developer can select slice criteria that, if removed, would have the largest impact on footprint reduction. Note that the act of actually removing a program slice requires an explicit action on the part of the developer (for instance, using the techniques in Section 2) and cannot be achieved by Shiv alone. However, for complex real-world programs with many dependencies, Shiv greatly simplifies the overall SCIRE technique. It prevents the developer from having to examine a large codebase method-by-method, looking for slice criteria. In addition, it can provide insights that may not be readily visible, such as the fact that the `println` statement has a dependency on Java's security module.

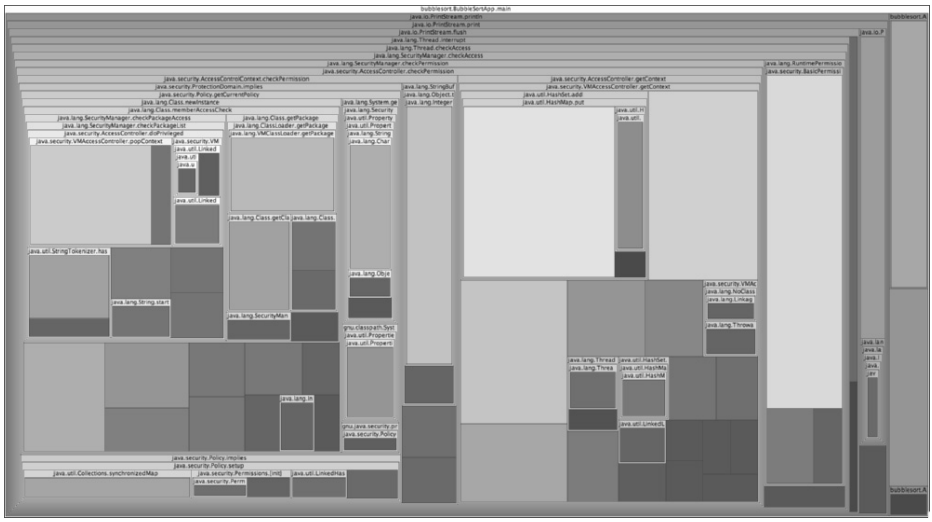


Figure 8. Shiv generates visualizations of embedded systems software, such as this treemap of the program in Figure 5. Each block represents the static code size of a program method: Larger, brighter blocks correspond to larger method sizes. Such visualizations help the developer easily identify methods that adversely affect code size.

4. EXPERIMENTAL RESULTS

This section presents the results of applying the SCIRE approach in conjunction with ProGuard,[1] a Java code shrinker, for reducing the static code size of JacORB 2.2,[2] a Java implementation of CORBA. For our measurements, the J2ME Connected Limited Device Configuration (CLDC) gives a reasonable target for code size: devices that have 160-512 KB of memory available for the Java platform and applications. Because the target size for these embedded devices is so small, even modest reductions in footprint become significant.

Our footprint measurements were based on the functionality required by two sample applications: 1) a *Simple Remote Client* and 2) *Supplier Dispatch using FACET*[3]. We collected the following measurements for each application: 1) the original size of JacORB before any reduction has taken place, 2) the size of the resulting library after code shrinking,

[1] http://proguard.sourceforge.net/
[2] http://www.jacorb.org/
[3] http://www.cs.wustl.edu/~doc/RandD/PCES/facet/

and 3) the size of the resulting library after applying both SCIRE and code shrinking.

Figure 9 shows that code shrinking alone provides a substantial reduction of code size: over 90% in relation to the original library. However, our proposed approach—the joint application of SCIRE and code shrinking—provides additional reduction: about 96% from the original size, and over 50% beyond the application of code shrinking alone. This additional reduction shows the effects of dependencies on unused features that remained in the code. Only the application of the SCIRE approach can enable the removal of such features. In fact, for both applications, SCIRE is necessary to bring the library size within the range specified by the CLDC.

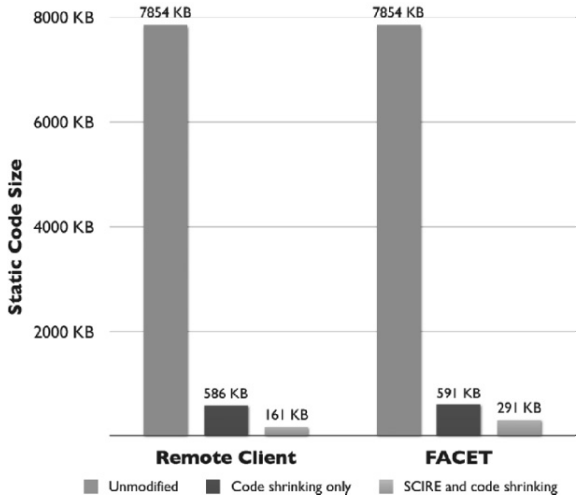


Figure 9. Results of footprint reduction in two sample applications: a CORBA remote client and a FACET event channel client.

REFERENCES

[Agrawal and Horgan, 1990] Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY.

[Baehrecke et al., 2004] Baehrecke, E. H., Dang, N., Babaria, K., and Shneiderman, B. (2004). Visualization and analysis of microarray and gene ontology data with treemaps. *BMC Bioinformatics*, 5(84).

[Johnson and Shneiderman, 1991] Johnson, B. and Shneiderman, B. (1991). Treemaps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization*, pages 284–291, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Weiser, 1981] Weiser, M. (1981). Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA. IEEE Press.

[Zhang et al., 2003] Zhang, X., Gupta, R., and Zhang, Y. (2003). Precise dynamic slicing algorithms.

CONFIGURABLE HYBRIDKERNEL FOR EMBEDDED REAL-TIME SYSTEMS

Timo Kerstan, Simon Oberthür
Heinz Nixdorf Institute, University Paderborn
Fürstenallee 11, 33102 Paderborn, Germany
timo.kerstan@uni-paderborn.de, zottel@uni-paderborn.de

Abstract: When designing a kernel for an operating system the developer has to choose between an microkernel or monolithic kernel approach. Bases for the decision is mostly the tradeoff between security and performance. Depending on application demands and on the available hardware a microkernel or a monolithic kernel approach or something between is desired. In this paper we present a hybrid kernel for embedded real-time systems which can be configured to the application demands in an easy way. To realize the hybrid kernel we present a technique to guarantee memory access in *O(1)* with virtual memory. With our approach the same codebase can be used for system services to be placed either in userspace or in kernelspace.

Keywords: real-time operating system, microkernel, monolithic kernel, virtual memory, memory management unit, MMU, translation look-aside buffer, TLB

1. INTRODUCTION

The operating range of embedded systems is varying from simple toys up to aircrafts. Hence real-time, performance and security demands are varying from negligible to indispensable. On the other hand most security features implemented in modern real-time operating systems [5] are at the expenses of performance, if not implemented or supported in dedicated hardware. A usual approach to increase security is application/operating system separation in embedded systems by using a kernel architecture. To increase security of service of the operating system, the services can be separated from basic operating system functionality as well. This is implemented by placing each service in userspace in a own separated address space.

When using the kernel approach the question is which functionality to keep in the kernel and which functionality to put in the userspace. As motivated from the security point of view putting as much as possible functionality in separated userspace sections is desired. From the performance point of view

Kerstan, T., Oberthür, S, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 279–288.

putting as little functionality in separated userspace sections is desired, to avoid expensive context switches and communication between different address spaces. Which direction of this tradeoff to choice depends highly on the applications demands and scenario.

In this paper, we present an extension of our fine granular configurable RTOS DREAMS [2]. We introduce a flexible configurable kernel implementation: A configurable Hybrid Kernel. Our approach allows an efficient way to configure which service to place in kernel and which in userspace. The main goal is to give the developer the possibility to adapt the kernel to his special needs regarding fault isolation for security and speed for efficiency.

To make system services in userspace as fast as possible high-performance context switching is crucial. High-performance context switch requires hardware specific tweaks. This paper presents an approach for our operating system on the PowePC405 architecture.

We first present the related work (2) which inspired our approach. The offline configurabilty of our approch is realized by the Skeleton Customization Language (SCL) (3.1). After presenting our hybridkernel (3.2), we shortly discuss an approach to minimize context switching time on our hardware architecture and to guarantee memory access in *O(1)* using virtual memory (3.3).

2. RELATED WORK

In the field of operating systems the pros and cons of microkernel vs. monolithic kernel approaches have been largely discussed (eg. [1], [8]). Looking at the market leading operating systems and real-time operating systems today both kernel designs can be found. The benefit of the microkernel approach is the isolation of operating system components in userspace. In the microkernel only basic and fundamental functions are implemented other features are separated in different address spaces. This increases fault tolerance, because bugs can only affect a small area of other code. In a big monolithic kernel a bug can effect on the whole functionality in the huge kernel. A small microkernel minimizes the number of bugs in the critical kernel routines because lines of code and number of bugs is related to each other (cp. [7]). The design paradigm of micro kernels (component seperation/development and standardized interaction model) implies a more structured design. Of course such a design can likewise be used in a monolithic kernel. The most mentioned benefit of monolithic kernel is the performance, because in microkernel each context switch is an overhead. The performance of micro kernel implementations can be massively improved by optimizing the kernel to special hardware. A technique to minimize context switch time is shown in this paper in section 3.3 or from Hildebrand in QNX [3].

Kea

Kea [9, 10] was designed for experimentation with kernel structures. It allows a fine-grained decomposition of system services and a dynamic migration of services between different adress spaces (Domains in Kea) during runtime. This includes the migration of services between userspace and kernelspace to accelerate frequently used services. To provide transparency for the developer and the system inter-domain-calls (IDCs) are introduced. These IDCs make use of so called portals which map the function call to the appropriate domain and service providing this function. This is done in a dynamic manner as service migration is possible during runtime. For real-time operating system this dynamic behaviour implies not deterministic latencies of function calls. Nevertheless the basic idea of a hybrid kernel providing safety, modularity and performance depending on the system requirements is the same as stated in this paper. This paper additionally focuses on embedded and real-time constraints.

Emeralds

EMERALDS[1] [11, 12] was developed at the University of Michigan and is a scientific prototype achieving an efficient microkernel for embedded systems with low resources. The main goal is to provide extremely fast system calls which are comparable to regular function calls. This shall expunge the disadvantage of slow system calls in microkernel architectures compared to monolithic kernel architectures.

To accelerate the system calls, EMERALDS always maps the kernel into every user process. The section of the kernel is protected against misuse of the user process. The advantage of this mapping is that at every system call no switch of the address space is necessary anymore reducing the complexity of the context switch from a user adress space to the kernel adress space. Due to the fact that no exchange of the virtual mappings has to be done, every pointer keeps its validity preventing the relocation of pointers and the data transfer between userspace and kernelspace. The developers of EMERALDS showed that the resulting complexity of a system call in EMERALDS is in the same dimension as a regular function call.

We need to note that EMERALDS is not really a microkernel as most of the services are realized in the kernelspace. This leads to a lack of security as an erroneous service may cause the whole system to crash.

[1] Extensible Microkernel for Embedded ReAL-time Distributed Systems

3. A CONFIGURABLE HYBRIDKERNEL FOR EMBEDDED REAL-TIME SYSTEMS

The configurable hybridkernel presented is based on the operating system DREAMS which is written in C++. The kernel is build on the minimal component ZeroDREAMS, is offline configurable and uses virtual memory to separate processes into different address spaces. The offline configurability is based on our Skeleton Customization Language (SCL).

3.1 Skeleton Customization Language

SCL [2] is used for the configuration of our hybrid kernel architecture. The SCL is a framework for offline fine granular source code configurability and was introduced together with the development of DREAMS. The main aspects provided by SCL are the configurability of the superclass and the members of a C++ class. The framework uses a configuration file to generate code out of the configuration file and the assigned source files.

To configure the *superclass* of a *subclass* the following expression is used

```
SKELETON SubClass IS A [VIRTUAL] path/SuperClass[(args)];}
```

The keyword *VIRTUAL* is optional and instructs the SCL framework to generate virtual methods. It is possible to specify the constructor of the superclass to be used for initialization by providing the parameters of the corresponding constructor.

To add a configurable member object to a class the developer has to provide the following line for each configurable member object:

```
MemberName ::= [NONE|DYNAMIC] path/ClassName[(args)];
```

That allows to configure the class of the member object and the constructor to be used. One of the key features here are the optional keywords *NONE* and *DYNAMIC*. The keyword *NONE* allows to define the abscence of this member object and the keyword *DYNAMIC* states that the member object is initialized when the first access occurs.

Because of the possibility to declare members absent the developer needs to consider this when writing his code. Therefore SCL creates managing methods for every configurable Member:

```
hasMemberName();
getMemberName();
setMemberName();
```

A short example shows how the resulting code would look like after the SCL Framework processed it. The example is used in a Single-Threading Environment in which no scheduling is necessary. This abscence of the scheduler is

described by the keyword *NONE*. The used architecture accesses the memory through a MMU (Memory Management Unit) and therefore a member object for the configuration of this MMU is available.

```
SKELETON PowerPC405 IS A Processor{
    MMU ::= path/PowerPC405MMU;
    Scheduler ::= NONE path/Scheduler
}

class PowerPC405 : Processor{
    ...
    hasMMU(){return true;};
    getMMU(){return radio;};
    setMMU(PowerPC405MMU arg){mmu = arg;};
    ...
    hasScheduler(){return false;};
    getScheduler(){return NULL;};
    setScheduler(){};
    ...
};
```

3.2 Hybridkernel

Using SCL allows a fine granular configurability within the source code of our hybridkernel. This configurability is used in the so called *Syscall Dispatcher*, which is a demultiplexer for system calls. It delegates the system calls to a configured service object. The handlers are realized as configurable member objects of the *Syscall Dispatcher* and can easily be exchanged within the SCL configuration. This allows to create an offline configurable kernel.

Our hybridkernel is able to place the services of our OS inside the kernelspace or in the userspace depending on the developers needs for security and performance. Therefore we are using proxies of the services which encapsulate only the communication, provided by the mikrokernel architecture (e.g. message passing). These proxies delegate the system calls to the appropriate service skeleton within the userspace and need to ensure that the semantics of the system calls are realized correctly (e.g. blocking system calls). An example configuration for a service moved from kernelspace to userspace is depicted in Figure 1. It is important to state that the proxy and the skeleton can be generated using state of the art techniques as they are used i.e. in Java RMI. If a service depends on another service we need to add a proxy for this service in the userspace variation. Also a skeleton to the kernelspace has to be added as an endpoint for the proxy.

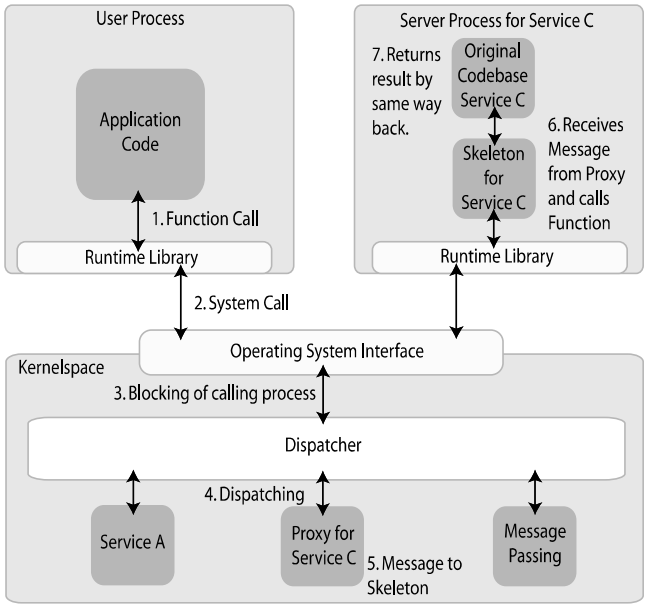


Figure 1. Migration of Service C from Kernelspace to Userspace

Using the regular scheduling mechanisms of a microkernel, the process containing the service has to wait until the scheduler activates it. This can lead to not predictable delay times which are depending on the used scheduling algorithm. In Real-Time Systems this delay needs to be predictable. To achieve this it is possible to use priority inheritance. The server process containig the services executing the system call, inherits the priority of the calling process. This leads to an immediate activation of the server process if no other process with higher priority is available. When the service is not needed a minimum priority can be assigned to the server process leading to an on demand activation of the service.

3.3 Context Switch and Virtual Memory access time

The microkernel architecture offers the possibility to separate the core of the OS (kernel) and services to provide a higher security by fault isolation. This separation comes at the cost of performance through complex context switches. Especially in embedded system the performance decreases rapidly when the context switch is not efficient and the applications make heavily use of system calls as the embedded systems are very restricted in ressources as computing power or memory. In our hybrid microkernel we want to minimize that tradeoff between security and performance. Therefore we use a similar approach as in

EMERALDS to realize an efficient context switch in a microkernel architecture and focus on the memory access time when using virtual memory in real-time systems.

EMERALDS enhances the context switch between a process and the kernel when a system call needs to be executed. Therefore the Kernel is mapped into the virtual address space of every process. That simplifies the context switch for a system call to the exchange of processor registers. Another advantage of that mechanism is that the kernel has direct access to the data of the calling process. This is very useful when a process wants to send data through a communication device. Nevertheless *EMERALDS* does not enhance the context switch between two processes. This leads to a big overhead in systems with multiple tasks where the time slices are pretty short. This is the reason why we also enhanced the context switch between processes.

To retain controll of the content of the TLB, we assume the use of a software managed TLB[2] for address translation inside the MMU[3]. When a context switch between two processes occurs the registers and the TLB needs to be filled with the entries of the next process. This can be very time consuming depending on the hardware architecture. In our case we tested our hybridkernel on a PowerPC 405 which has a software managed TLB with 64 entries. The time to restore the registers was determined to be about $1.5\mu s$ (see 3.4) and the time to write the 64 TLB entries was determined to be about $64\mu s$. This shows that the main overhead is caused by the writing of the TLB for the address space switch.

Another important issue in real-time systems is that every memory access could lead to a TLB Miss. TLB misses occur when there is no adress translation available in the TLB for the requested virtual adress. In that case the MMU has to replace an existing entry in the TLB with the needed entry from the virtual mapping of the process. In our case a TLB Miss causes a minimum overhead of about $1\mu s$ for writing the needed entry to the TLB. This overhead has to be considered in the WCET[4] and decreases the overall performance.

To guarantee an efficient context switch also between two process and a *O(1)* memory access time we make the restriction to the total number of TLB Entries needed by all processes (TLB_{system}) not to exceed the maximum number of TLB entries supported by the MMU (TLB_{MMU}). This guarantees that no TLB Entries need to be exchanged during a context switch and during runtime no TLB Miss can occur.

$$TLB_{system} < TLB_{MMU}$$

[2]Translation Look-Aside Buffer. TLBs are used to speed up the address translation process
[3]Memory Management Unit. The MMU translates virtual addresses into physical addresses using the TLB.
[4]Worst Case Execution Time. Needed for schedulability analysis in real-time systems

This also means that all processes of the complete system need to share the TLB. In systems with static page sizes this is pretty hard depending on the size of the pages (The smaller the harder). Some architectures offer the possibility to use memory segments with different sizes. The used PowerPC 405 MMU supports dynamic segments between 4K and 16MB. This allows to allocate a large block of contigous memory with only 1 TLB entry. To reduce the number of used TLB entries in the system the Intermediate-level Skip Multi-Size Paging algorithm [6] can be applied which is based on the well known buddy memory allocation algorithm [4].

3.4 Performance

As already stated the bottleneck of the microkernel architectures is the efficiency of the context switch which directly influences the performance of system calls. Therefore we determined the time for a NoOp system call by simply executing a large number of NoOp system calls from Userspace measuring the time until the last system call returns. The result of this is then divided by the number of systemcalls. As a result we get a mean execution time of $1.51\mu s$ for a NoOP system call on an Avnet Board with Virtex-II Pro (PowerPC405 with a clock of 200 MHz). A NoOp systemcall consists of two context switches. The mean execution time for a system call showed to be a good approximation of a single system call. We compared this mean execution time for a system call to the time of a simple NoOp function call within the userspace to show the overhead of the microkernel architecture compared to a library based OS. The needed time for a NoOp function call was $0.11\mu s$ beeing about 15 times faster than a system call. When converting this to cycles this is an overhead of about 280 cycles for a system call in comparison to a function call. If the service is located in the userspace two more context switches are necessary adding about another 280 cycles of overhead as we can assume that about 140 cycles are necessary for a single context switch.

Operation	Time	Clockcycles
NoOP Function Call (PowerPC405@200MHz)	$0.11\mu s$	22
NoOP System Call (PowerPC405@200MHz)	$1.51\mu s$	302

Table 1. Performance

Another thing to mention is the overhead through the configurability using SCL. We showed in section 3.1 that some code is inserted when configurable members are implemented. This code could lead to a small overhead every time the configurable member is accessed. In many cases this is not the case as the compilers today are pretty good in optimizing the code and removing code sections which are not necessary. Such an unnecessary code segment occur i.e.

in combination with the *hasMemberName()* method to check the availability of a configurable member. In the following example the presence of a scheduler is checked and the code would look like this:

```
if (hasScheduler()){
  getScheduler->schedule();
}
```

The compiler would remove the whole code if *hasScheduler()* returns false. Otherwise the resulting code would look like this

```
getScheduler->schedule();
```

4. CONCLUSION

We presented our concept of a Hybridkernel in which operating system components can be flexible configured in userspace or kernelspace. The advantage of this approach is that the same code base for the services can be used. Our well structured fine granular configurable operating system DREAMS allows an easy separation of system components with our configuration language SCL. This enables the application designer to choose the best tradeoff for his application between security and performance. Additionally we presented our fast concepts for high-performance context switches on the PowerPC405 architecture. Which enables the application designer to choice the more secure micro kernel approach for his applications.

The strength in security of the mikrokernel concept comes at the cost of performance. As we stated in this paper there is a limit in the capacity of the memory management units which lead to non deterministic memory access times if their capacity is exceeded. Our current approach will come to its limit if there is a lack of memory or the number of processes is growing. Therefore we will try to enhance the hardware support by using partially reconfigurable FPGAs. These FPGAs will then be used as an external MMU which will be highly adaptable to the needs of the system.

REFERENCES

[1] B. Behlendorf, S. Bradner, J. Hamerly, K. Mckusick, T. O'Reilly, T. Paquin, B. Perens, E. Raymond, R. Stallman, M. Tiemann, L. Torvalds, P. Vixie, L. Wall, and B. Young. *Open Sources: Voices from the Open Source Revolution*. O'Reilly, February 1999.

[2] C. Ditze. *Towards Operating System Synthesis*. Phd thesis, Department of Computer Science, Paderborn University, Paderborn, Germany, 1999.

[3] D. Hildebrand. A microkernel posix os for realtime embedded systems. In *Proceedings of the Embedded Computer Conference and Exposition 1993*, page 1601, Santa Clara, april 1993.

[4] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[5] J. A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Syst.*, 28(2-3):237–253, 2004.

[6] S. Suzuki and K. G. Shin. On memory protection in real-time os for small embedded systems. In *RTCSA '97: Proceedings of the 4th International Workshop on Real-Time Computing Systems and Applications (RTCSA '97)*, page 51, Washington, DC, USA, 1997. IEEE Computer Society.

[7] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[8] L. Torvalds. The linux edge. *Commun. ACM*, 42(4):38–39, 1999.

[9] A. Veitch and N. Hutchinson. Kea-a dynamically extensible and configurable operating system kernel. In *Configurable Distributed Systems, 1996. Proceedings., Third International Conference on*, pages 236–242, 6-8 May 1996.

[10] A. Veitch and N. Hutchinson. Dynamic service reconfiguration and migration in the kea kernel. In *Configurable Distributed Systems, 1998. Proceedings., Fourth International Conference on*, pages 156–163, 4-6 May 1998.

[11] K. Zuberi and K. Shin. Emeralds: a microkernel for embedded real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 241–249, 10-12 June 1996.

[12] K. Zuberi and K. Shin. Emeralds: a small-memory real-time microkernel. *Software Engineering, IEEE Transactions on*, 27(10):909–928, Oct. 2001.

EMBEDDED SOFTWARE DEVELOPMENT IN A SYSTEM-LEVEL DESIGN FLOW

Gunar Schirner, Gautam Sachdeva, Andreas Gerstlauer, Rainer Dömer
Center for Embedded Computer Systems, University of California Irvine
{hschirne, gsachdev, gerstl, doemer}@cecs.uci.edu

Abstract: System level design is considered a major approach to tackle the complexity of modern System-on-Chip designs. Embedded software within SoCs is gaining importance as it addresses the increasing need for flexible and feature-rich solutions. Therefore, integrating software design and co-simulation into a system level design flow is highly desirable.

In this article, we present the software perspective within our system-level design flow. We address three major aspects: (1) modeling of a processor (from abstract to ISS-based), (2) porting of an RTOS, and (3) the embedded software generation including RTOS targeting.

We describe these aspects based on a case study for the ARM7TDMI processor. We show processor models including a cycle-accurate ISS-based model (using SWARM), which executes the RTOS MicroC/OS-II. We demonstrate our flow with an automotive application of anti-lock breaks using one ECU and CAN-connected sensors. Our experimental results show that automatic SW generation is achievable and that SW designers can utilize the system level benefits. This allows the designer to develop applications more efficiently at the abstract system level.

Keywords: Embedded Software Development, System Level Design, TLM

1. INTRODUCTION

Embedded software plays an important role in todays complex SoCs since it allows to flexibly realize a large feature set. However, writing software manually is not desirable due to the amount of code and the hardware often not being available in early stages. Therefore, it is highly desirable to address software development as early as possible. To accelerate the design process and to increase the productivity, system-level design has to accommodate software concerns enabling a seamless co-design of software and hardware.

Schirner, G., Sachdeva, G., Gerstlauer, A., Dömer, R, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 289–298.

1.1 Problem Statement

In order to reduce the time-to-market, designers utilize system-level design that reduces the complexity by moving to higher levels of abstraction. Time and cost of software development can be dramatically reduced when properly integrated into the system-level design flow.

In this article, we describe software aspects in our system-level design flow [1]. We focus on three major elements (see Figure 1) that are crucial to the software support:

- Processor models at different levels of abstraction.
- Real-Time Operating System (RTOS) support.
- Generation of software, targeted to a selected RTOS.

This article describes each element based on a case study for an ARM7TDMI microprocessor.

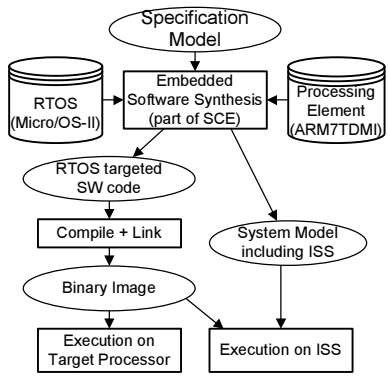


Figure 1. Software generation within system level design flow.

1.2 Related Work

System-level modeling has become an important issue, as a means to improve the SoC design process. System Level Design Languages (SLDLs) for capturing such models have been developed (e.g. SystemC [15], SpecC [10]). Significant research effort has been invested into frameworks for system level design and software synthesis.

Benini et al. [5] introduce MPARM, an MPSoC platform simulator. It provides a multi-processor cycle-accurate architectural simulator by encapsulating different Instruction Set Simulators (ISSs) in SystemC. Its main purpose is the analysis and profiling of system performance. Our approach, on the other hand, focuses on the generation of software.

Herrara et al. [17] describe an approach for embedded software generation from SystemC that overloads SystemC class library elements in order to use the same code for simulation and target execution. However, the approach imposes strict requirements to the input specification.

Several commercial tool sets provide integrated simulation environments for SoC designs: e.g. ARM's SoC Designer [3] and CoWare's Virtual Platform Designer [7]. While these tools focus on the ISS-based co-simulation aspect, our approach additionally includes abstract processor modeling and software generation.

In previous work [24], we describe abstract simulation models for processors. Similar work includes [11]. In this paper, we focus on the design flow to automatically generate the target software.

1.3 Outline

This document is organized as follows: Section 2.1 describes modeling of the ARM7TDMI [4] processor from abstract to ISS-based models. Section 2.2 reports on the RTOS support for the selected processor. In Section 2.3, we give an overview of the software generation and targeting. Finally, we demonstrate the resulting integrated flow in Section 3.

2. SOFTWARE SUPPORT

Supporting software development in system level design requires three major elements: processor modeling, RTOS support and generation of the embedded software.

2.1 Processor Modeling

Processor modeling captures an existing processor at different levels of abstraction, describing its characteristics and behavior [12].

For this case study, we chose the ARM7TDMI [4], a widely used 32-bit embedded RISC microprocessor [2]. It uses a three-stage pipeline (fetch, decode and execute) and has single a 32-bit bus interface carrying both instructions and data. The ARM7TDMI has two level-sensitive interrupts (nIRQ, nFIQ). Using the AMBA Design Kit [2], the ARM7TDMI connects to the Advanced High-performance Bus (AHB).

We have captured the processor at different levels of abstraction, starting with the most abstract behavioral model, then the bus functional model, and finally the ISS-based cycle-accurate model.

2.1.1 Behavioral Model. The behavioral model is the most abstract representation of the processor capturing only basic characteristics. It enables performance analysis in the early stages of the design.

The basic characteristics include the clock frequency, computing power in MIPS, power consumption, instruction width, data width, and data and program memory size. Some attributes are specified as ranges, for adaptation to the particular design needs (e.g. clock frequency).

Weight tables [6] constitute the main portion of the behavioral model. They are used for interpretation of generic profiling results and yield comparative analysis of different design alternatives. The ARM7TDMI behavioral model contains two weight tables, one for execution speed and the other for footprint estimation.

The computation weight table correlates C-level operations with clock cycles needed for their execution. It contains one entry for each operation and data type stating the minimal number of execution cycles. Using this table and the profiling information for each basic block, the system performance can be estimated, giving an early performance comparison between designs satisfying the fidelity property [9]. Similarly, a second weight table contains parameters for estimating the code size.

2.1.2 Bus Functional Model. The Bus Functional Model (BFM) is a pin-accurate and cycle-approximate model describing the processor's communication interfaces and behavior.

As shown in Figure 2, the ARM7TDMI BFM consists of a behavior hierarchy. The outer shell contains three parallel executing behaviors: processor core, Programmable Interrupt Controller (PIC) and timer. The core consists of two parallel executing behaviors: Hardware Abstraction Layer (HAL) and Interrupt Request (IRQ). The initially empty HAL will contain the user computation behaviors, which get inserted in the design flow. The IRQ behavior houses the logic for interrupt detection and handling. Upon receiving a core interrupt, it preempts execution of user code in the core and starts the system interrupt handler.

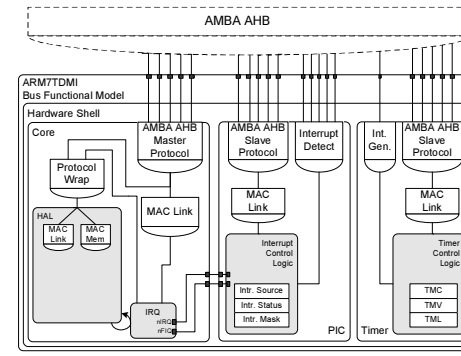


Figure 2. ARM7TDMI BFM.

The core communicates through the AMBA AHB master interface with external components (e.g. PIC and timer). The PIC [21] maps 32 external interrupts (each maskable and configurable in priority) to the core interrupts (nIRQ, nFIQ). The timer, disabled by default, generates periodic interrupts needed for the timing services of the RTOS.

The bus interface is realized as a layered set of inlined channels [23]. The additional channel *Protocol Wrap* disables interrupts during a bus transfer by the processor core to maintain accurate protocol timing.

2.1.3 Cycle-Accurate Processor Model. Accurate simulation of the software is offered by the cycle-accurate (CA) processor model based on an ISS integration, therefore also called instruction set model, and allows the execution of the final target binaries to validate the software synthesis output (Section 2.3).

We integrated the ISS SoftWareARM (SWARM) [8] into our model. SWARM provides cycle-accurate simulation of the ARM data path, includes a cache model and 12MB internal memory. SWARM additionally includes peripherals for PIC, timer, LCD and UART controller [18].

Our CA model (Figure 3) contains the *Core_ISS* behavior, which replaces the *Core* behavior of the BFM. Inside the *Core_ISS*, the behavior *ARM7TDMI_ISS* wraps the SWARM ISS and calls it cycle-by-cycle. The wrapping behavior interfaces with the remaining design through bus accesses and interrupts. It detects SWARM external memory accesses and executes them using the AMBA AHB master interface. It monitors the interrupt inputs (nIRQ, nFIQ) and triggers an ISS-internal interrupt using the SWARM API. The wrapping behavior advances the system's simulated time according to the ARM7 clock definition.

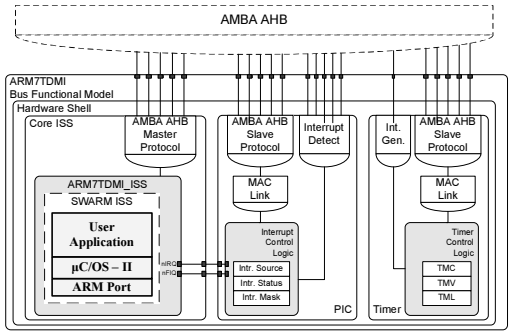


Figure 3. ARM7TDMI cycle-accurate model.

We disabled the SWARM internal PIC, timer and UART. Instead, we reuse the PIC and timer of the BFM, which communicate through the AHB. Upon startup, SWARM loads the target binary into the SWARM internal memory (address zero), where the execution then starts from.

2.2 Real-Time Operating System

The designer may, in the refinement process, assign multiple behaviors to one processor. Due to the inherently sequential execution, behaviors then have to be either statically or dynamically scheduled. An RTOS is needed on the target processor for dynamic scheduling.

We chose the μC/OS-II [19], which is a real-time kernel providing preemptive priority-based scheduling for up to 56 tasks. It offers deterministic services for inter-task communication, synchronization and timing. μC/OS-II, mostly implemented in ANSI C, is highly configurable to reduce the footprint (down to 2K bytes [19]).

The RTOS requires porting to execute on top of the SWARM ISS (see Figure 3). We based our processor adaptation on an available ARM port [20], and adjusted for the gcc cross-compiler [14] in terms of stack layout, data sizes and assembly code. We adapted context switch, interrupt handling and timing functions.

We use a two stage approach for the interrupt handling. The first stage handler, *OS_CPU_IRQ_ISR()*, is implemented in assembly. It saves the context of the current task onto the stack, calls the second stage handler, executes the OS scheduler and restores the new task's context. Its code is processor, compiler and OS dependent.

The second stage interrupt handler, implemented in C, performs the communication with the PIC to determine and clear the interrupt in the PIC, and then calls the user interrupt service routine. We chose this multi-stage approach to limit the amount of highly specialized code.

The RTOS relies on a timer interrupt to provide timing services. Our Hardware Abstraction Layer (HAL) contains a timer driver that configures the timer. We chose a 10ms period trading off between timing granularity and execution overhead.

2.3 Embedded Software Generation

The embedded software generation creates the final software implementation based on the design models. The generation process is embedded into the system refinement flow as introduced in Figure 1.

The overall flow starts with an abstract specification model captured in the SLDL SpecC [10]. Through step-wise refinement, the designer adds design decisions and explores different alternatives. Decisions include the allocation of processing elements (PEs), mapping of computation to the PEs, selecting scheduling parameters, and defining communication parameters including the bus mapping.

One refinement output is a BFM of the entire system. In this model, computation is mapped to processing elements [22], computation within a processor is grouped to tasks (with priorities) [13], communication is refined to bus primitives, and external synchronization is implemented (e.g. polling or interrupt) based on the designer's choice [25].

Our embedded software generation uses the BFM as an input and is divided into the C-code synthesis and the RTOS targeting.

2.3.1 C-Code Synthesis. The C-code synthesis is based on [26] and translates the SLDL statements into C-code. It resolves the behavioral hierarchy, behavior local variables and the port mappings to C constructs using functions, structures and pointers.

2.3.2 RTOS Targeting. RTOS targeting adapts the C-code for execution on the target processor, scheduled by an RTOS. We use a thin adapter, the RTOS Abstraction Layer (RAL). The RAL abstracts from the actual RTOS implementation, providing a canonical interface. RTOS targeting replaces SLDL statements for parallel execution into

calls to the RAL. It also adapts intra-processor communication to use RAL services. Figure 4 shows the resulting software stack.

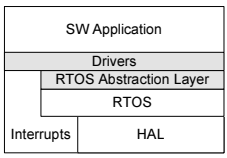


Figure 4. SW stack.

Additionally, RTOS targeting realizes external communication and synchronization. Most of the external communication has already been refined by earlier refinement steps [25] to use a canonical Media Access Control (MAC) layer, and can be translated automatically by the C-code synthesis. RTOS targeting only introduces a bus and processor-specific MAC layer implementation.

In case of interrupt based synchronization, RTOS targeting extracts the interrupt handlers from the simulation behaviors and creates user interrupt handlers. It generates startup code that initializes the RTOS and registers the user interrupt handlers to the system interrupt handler.

We have extended the refinement environment's database segregating the software subsystems by dependencies. This allows RTOS targeting to flexibly compose the final binary while minimizing code duplication in the database. We added the RTOS, RTOS-specific RAL, the RTOS port, and the board-specific HAL (containing PIC, timer and MAC code).

In the final step, the generated code is cross-compiled using gcc [14] and linked against the target and RTOS-specific libraries. This produces the final target binary, ready for execution on the target processor.

3. EXPERIMENTAL RESULTS

In order to show the feasibility of our design flow, we applied it to an example from the automotive domain. We implemented an anti-lock break system (see Figure 5). It uses one Electronic Control Unit (ECU) containing an ARM7TDMI, which executes the control application, and a transducer connecting to the Controller Area Network (CAN) [16]. Five sensors and actuators are connected through the CAN bus measuring the break paddle position, wheel speed and control the break pressure valve.

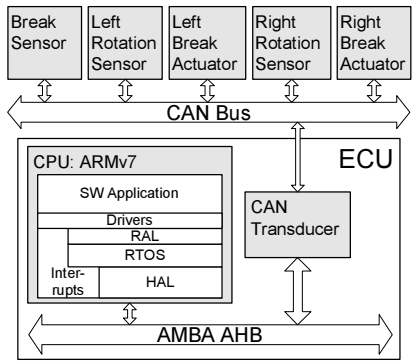


Figure 5. Anti-lock break example.

We captured this application as a specification model in SpecC [10] and used the automatic refinement to generate the BFM inserting design decisions that yield the desired target architecture using the ear-

lier described processor models. We then synthesised targeted C-code using the extended software generation (see Section 2.3), creating an executable binary for the ARM7TDMI processor. Using the ISS-based model (Section 2.1.3), we co-simulate the complete system. This allows us to validate the system and to analyze its performance in detail.

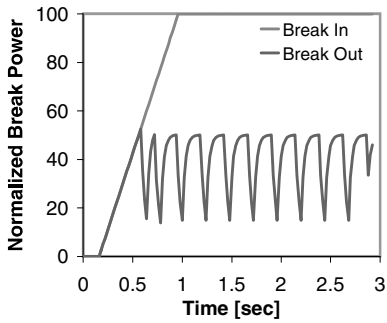


Figure 6. Anti-lock break simulation.

We validated correct functional execution of all created models using a simulated emergency stop maneuver with an initial speed of 20 $\frac{meters}{second}$ (45mph, 72$\frac{km}{h}$). Figure 6 shows the correlation between the break request (*Break In*), as read from the break paddle, and the computed break pressure *Break Out* asserted by the left valve actuator. As the requested break pressure increases, the left wheel loses traction in this scenario. Then, the anti-lock algorithm limits the asserted break power until sufficient traction is achieved, at which point the pressure is increased again.

Please note that we automated the refinement process and scripted the decision input. This allows us to develop code at the specification level while all other models are generated automatically. As a result, we did not have to deal with low-level implementation details and could focus on algorithm design at a higher level, gaining productivity. Furthermore, the automated refinement process allows us to generate all models within minutes, which enables design space exploration (although not shown for this example) and quick implementation turn around.

To give an indicator of model complexity, we measured execution times and lines of generated code. Table 1 summarizes the results which show that the execution time is negligible for the abstract models up to the TLM [24]. Starting with the BFM, the execution time dramatically increases exceeding two hours. This demo application is not computation intense and computation is spread over time. Most simulation effort is spent on the idle bus systems. Both, the AHB (running at 25 MHz) and the CAN (1 MHz) use explicit clocks. Especially the CAN contributes to a steep

Model	Lines of Code	Simulation Time
Spec	238	0.018sec
TLM	22035	0.153sec
BFM	22048	125min
BFM(ISS)/C	22390/1416	208min

Table 1. Model complexity in lines of code and simulation time.

performance drop with a local clock in each node[1]. The ISS-based model is 66% slower than the BFM. Simulating the ARM (25 Mhz) cycle-by-cycle adds a significant overhead. Again, with the minimal computation, the processor spends most cycles in the idle task.

Analyzing the lines of code shows that adding the abstract models (for processor, bus and hardware components) significantly increases the size showing the value of automatic model generation. Also, the final user code of 1416 lines C code is much larger than the initial specification model. Significant detail has been added throughout the generation process for communication, task control and boiler plate code.

4. CONCLUSIONS

In this article, we have presented the software perspective of our system-level design flow. In form of a ARM7TDMI based case study, we described three major tasks necessary for software support in our flow.

First, we described the processor modeling at different abstraction levels with the behavioral model for early exploration as our most abstract model. We reported on the pin-accurate BFM and furthermore showed the successful integration of an ARM7 ISS into a software cycle-accurate model. Second, we discussed the adaptation of the μC/OS-II to the ARM7 processor. Third, we reported on the embedded software synthesis describing the RTOS targeting extension. We generate C code for the selected RTOS and support internal communication, external communication and synchronization.

Using an anti-lock break example, we have demonstrated the design flow utilizing the three introduced SW support tasks. All generated models, including the ISS-based model, simulate correctly, validating our ability to automatically generate the final software implementation. Using our automated refinement flow, a software developer will benefit from describing the application at a higher level, where communication details, processor specifics and RTOS API are hidden. Yet, the flow produces the detailed implementation within minutes.

REFERENCES

[1] S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Dömer, and D. Gajski. System-on-Chip Environment (SCE Version 2.2.0 Beta): Tutorial. Technical Report CECS-TR-03-41, CECS, University of California, Irvine, July 2003.

[2] Advanced RISC Machines Ltd. (ARM). ARM7TDMI (Rev 3) Product Overview. www.arm.com/pdfs/DVI0027B_7_R3.pdf.

[1] Note that each bit on the CAN bus is oversampled (e.g. 12 times, [16]) for synchronization with the sending node. Thus, a higher frequency is needed for each local clock.

[3] Advanced RISC Machines Ltd. (ARM). SoC Developer with MaxSim Technology. www.arm.com/products/DevTools/MaxSim.html.

[4] Advanced RISC Machines Ltd. (ARM). ARM7TDMI (Rev 4) Technical Reference Manual, 2001. www.arm.com/pdfs/DDI0210B7TDMIR4.pdf.

[5] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivier. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *VLSI Signal Processing*, 41:169–182, 2005.

[6] L. Cai, A. Gerstlauer, and D. D. Gajski. Retargetable Profiling for Rapid, Early System-Level Design Space Exploration. In *DAC*, San Diego, CA, June 2004.

[7] CoWare. Virtual Platform Designer. www.coware.com.

[8] M. Dales. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow, Nov. 2000. www.cl.cam.ac.uk/~mwd24/phd/swarm.html.

[9] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[10] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[11] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. A. Jerraya. Flexible and Executable Hardware/Software Interface Modeling for Multiprocessor SoC Design Using SystemC. In *ASPDAC*, Yokohama, Japan, Jan. 2007.

[12] A. Gerstlauer, G. Schirner, D. Shin, J. Peng, R. Dömer, and D. D. Gajski. System-On-Chip Component Models. Technical Report CECS-TR-06-10, Center for Embedded Computer Systems, University of California, Irvine, May 2006.

[13] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS Modeling for System Level Design. In *DATE*, Munich, Germany, March 2003.

[14] GNU. gcc (gcc-arm-coff version 2.95.3). ftp://ftp.gnu.org/gnu/gcc.

[15] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[16] F. Hartwich and A. Bassemir. The Configuration of the CAN Bit Timing. www.can.bosch.com, 1999.

[17] F. Herrera, H. Posadas, P. Snchez, and E. Villar. Systematic Embedded Software Generation from SystemC. In *DATE*, 2003.

[18] Intel Corporation. Intel StrongARM SA-1110 Microporcessor Developer's Manual. developer.intel.com/design/strong/manuals/278240.htm, October 2001.

[19] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 2002.

[20] Micriμm. *μC/OS-II and The ARM Processor, Application Note AN-1011*, 2004.

[21] NEC Electronics (Europe) GmbH. System-on-Chip Lite +. User's Manual. www.eu.necel.com/_pdf/A17158EE2V0UM00.PDF, April 2005.

[22] J. Peng, S. Abdi, and D. D. Gajski. Automatic Model Refinement for Fast Architecture Exploration. In *ASPDAC*, Bangalore, India, January 2002.

[23] G. Schirner and R. Dömer. Quantitative Analysis of Transaction Level Models for the AMBA Bus. In *DATE*, Munich, Germany, March 2006.

[24] G. Schirner, A. Gerstlauer, and R. Doemer. Abstract, Multifaceted Modeling of Embedded Processors for System Level Design. In *ASPDAC*, Yokohama, Japan, Jan. 2007.

[25] D. Shin, A. Gerstlauer, J. Peng, R. Dömer, and D. D. Gajski. Automatic Generation of Transaction-Level Models for Rapid Design Space Exploration. In *CODES+ISSS*, Seoul, Korea, Oct. 2006.

[26] H. Yu, R. Dömer, and D. D. Gajski. Embedded Software Generation from System Level Design Languages. In *ASPDAC*, Yokohama, Japan, January 2004.

DATA REUSE DRIVEN MEMORY AND NETWORK-ON-CHIP CO-SYNTHESIS*

Ilya Issenin and Nikil Dutt
University of California, Irvine, CA 92697

Abstract: NoCs present a possible communication infrastructure solution to deal with increased design complexity and shrinking time-to-market. The communication infrastructure is a significant source of energy consumption and many attempts at energy efficient NoC synthesis have been proposed. However, in addition to the communication subsystem, the memory subsystem is an important contributor to chip energy consumption. These two subsystems are not independent, and a design with the lowest memory power consumption may not have the lowest overall power consumption. In this paper we propose to exploit a data reuse analysis approach for co-synthesis of memory and NoC communication architectures. We present a co-synthesis heuristic targeting NoCs, such as Æthereal, with mesh topology. We show that our data reuse analysis based synthesis reduces communication energy alone by 31% on average as well as memory and communication energy by 44% on average in comparison with the similar approaches that do not employ data reuse analysis. We also show that our memory/NoC co-synthesis heuristic further reduces communication energy by up to 38% in comparison with a computationally less expensive traditional two-step synthesis approach, where the memory subsystem is synthesized before the synthesis of NoC. To the best of our knowledge, this is the first work to investigate the influence of memory subsystem synthesis on NoC power consumption and to present a memory and NoC co-synthesis heuristic.

Key words: Network-on-Chip synthesis, memory synthesis, data reuse analysis

1. INTRODUCTION

Multiprocessor Systems-on-Chips (MPSoCs) are becoming a popular solution to meet the growing processing demands of embedded applications.

* This work was partially supported by NSF grant CCR-0203813.

Issenin, I., Dutt, N, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 299–312.

One important aspect of SoC design is the communication subsystem. There are several conflicting requirements for this subsystem: it should be low power, scale well with increasing SoC size and allow easy modifications of the chip when part of the chip has to be redesigned. Due to good scalability, Network-on-Chip (NoC) is one of the promising communication architectures that is well suited for larger SoCs.

Memory and communication subsystem are both important contributors to SoC energy consumption, which is a scarce resource in typical application of embedded SoCs. While there are a number of approaches for minimizing energy consumption in memories and NoCs, traditionally such approaches are applied independently, typically with the memory subsystem designed first, followed by NoC synthesis. This may result in a system that doesn't have the lowest possible overall power consumption. Our work focuses on co-synthesis of NoC together with a memory architecture. We obtain high customization of memory subsystem by employing our data reuse analysis technique DRDM [10]. To the best of our knowledge, this is the first work to explore the influence of memory subsystem configuration on NoC power consumption and to propose a co-synthesis heuristic aimed at minimizing total power consumption.

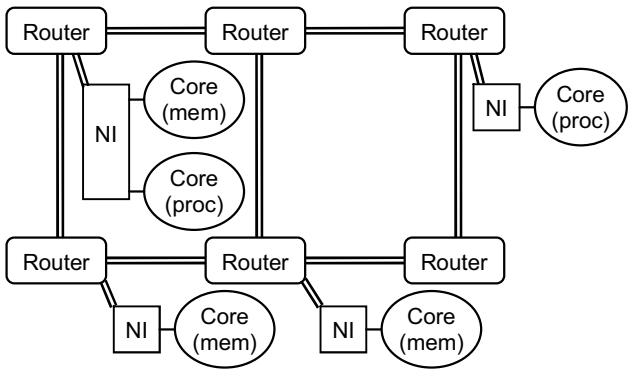


Figure 1. Example of 3x2 mesh-type NoC communication architecture

To illustrate our approach we target the Æthereal NoC [3][5], although our approach is applicable to other NoCs as well. Æthereal is a synchronous NoC that provides guaranteed throughput, fixed latency communication between cores. Figure 1 shows a typical NoC architecture with 3x2 mesh topology. The system consists of communicating cores and routers with network interfaces (NIs) connected by links. The routers provide packet routing according to a predetermined static schedule. Network interfaces provide conversion of packed-based NoC communication to protocols that cores use, as well as clock domains conversion [3]. The cores represent

processors, memories with or without DMA controllers, or I/O devices. For instance, the NoC depicted in Figure 1 has two cores that are processors and three cores that contain memory.

The problem of NoC co-synthesis is the task of customizing the memory configuration together with determining which network interface each core is connected to, and identifying the routing of the packets between communicating network interfaces.

Traditionally, communication requirements between cores (which are used as input information for NoC synthesis) are represented by communication graph, which consists of nodes representing communicating cores and edges that specify the direction and amount of communication between the cores. This is also called communication throughput graph in [17], core graph in [16] or application characterization graph in [9]. Such a representation implies that the number of cores (processors, memories, peripheral devices) and communication requirements are fixed before the NoC synthesis.

In contrast, we do not fix the memory architecture before the NoC synthesis. In our approach we use a *data reuse graph* that concisely captures memory reuse patterns that can be exploited to perform effective memory customizations. It can be also viewed as a set of different communication graphs, which differ in the number of additional buffers that exploit data reuse (temporal locality) in processor accesses. The *data reuse graph* can be obtained by performing data reuse analysis [10].

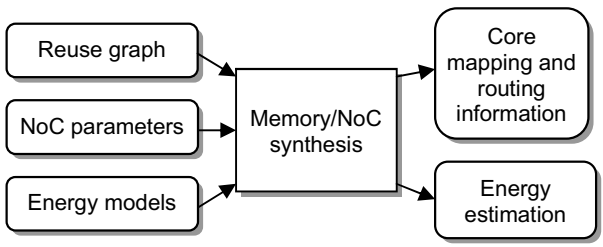


Figure 2. Synthesis framework

The design flow of our approach is depicted in Figure 2. It takes application reuse graph (obtained with our technique described in [10]), NoC parameters (such as mesh size) and energy models as input. After performing synthesis with one of the algorithms described later in Section 4, the framework provide synthesis information as well as estimation of the energy consumption of the synthesized configuration.

In this work we present a memory/NoC co-synthesis power-aware heuristic that uses the data reuse graph as input and compare it with a two-step heuristic, where the memory configuration is decided before the NoC

synthesis. We show that our proposed co-synthesis heuristic outperforms the two-step synthesis approach by achieving reductions of up to 38% in communication (NoC) energy and by up to 26% in combined communication and memory energy.

2. RELATED WORK

The potential of using NoCs in MPSoCs as a future on-chip interconnects has been recognized in [2][6][8]. Since then a large amount of research was done investigating different aspects of NoC, such as NoC architecture, routing, interconnect technology, development support, etc.

In the area of NoC synthesis, a number of researchers have proposed core mapping and routing techniques for NoC architectures. Murali et al. [16] present two techniques aimed at average communication delay reduction in the presence of bandwidth constraints. Srinivasan et al. [19] proposed technique that handles both bandwidth and latency constraints. Hu et al. [9] perform energy aware mapping with the bandwidth constraints. Manolache et al. [13] add redundancy in message transmission as a way to address the problem of transient link failures. Guz et al. [7] solve the problem of allocating link capacities in NoC with non-uniform link capacities. Srinivasan et al. [20] present layout aware mapping and routing approach.

In [19] it was shown that the NMAP algorithm from [16] provides solutions within 10% from the optimal ones obtained by MILP, and it is on par (or better on average) than the solutions obtained by algorithm from [19] if no latency constraints (only bandwidth constraints) are taken into account. We use a modified NMAP algorithm in our technique to perform NoC synthesis.

All of the abovementioned techniques work with a given communication graph and do not perform any memory customizations during the synthesis. The advantages of using memory/communication co-synthesis was shown in [11][17] for bus-based communication architectures. To the best of our knowledge, our approach is the first work that applies memory/communication co-synthesis to MPSoCs which use NoC interconnect architecture.

3. MEMORY MODEL AND DATA REUSE GRAPH

In our approach we employ *data reuse graphs* produced by a data reuse analysis technique [10]. They are built by analyzing the application's memory request patterns. Reuse graphs represent a hierarchy of buffers, with

each buffer holding the data that is used several times by the processors or by the buffers which are lower in the memory hierarchy. Each reuse graph node (or buffer) can be private, i.e., providing data to only one processor, or shared, holding data that are used by several processors. Reuse graph edges represent the data flow, i.e., shows the source or destination of the data that is kept in the buffer. An approach to modify the application code to include the necessary DMA transfers between the buffers and provide synchronization between updates of the shared buffers and processors is also included in [10].

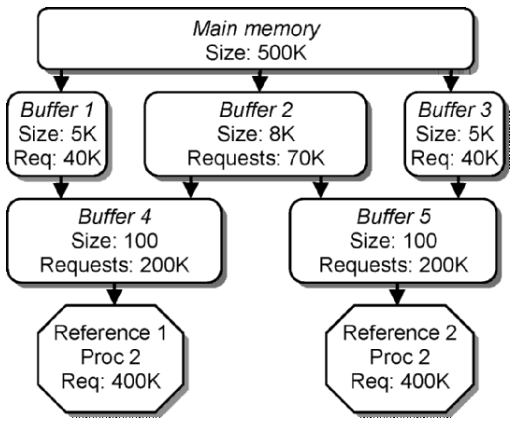


Figure 3. Example of a multiprocessor data reuse graph

An example of data reuse graph is shown in Figure 3. It has several private buffers (1, 3-5) and one shared buffer (2). The data reuse analysis tool provides the sizes of the buffers and the number of data requests to a memory of higher level in memory hierarchy.

Each reuse graph buffer may be mapped to a physical memory and implemented as a NoC core (possibly together with DMA engine). Implementing a buffer may reduce the overall energy consumption since it can be placed near data consuming core in a small and low energy consuming memory, thus eliminating repeated memory accesses to more energy expensive main memory. However, in other cases, it may lead to an increase in energy consumption due to the overheads of copying data from one memory to another and additional power consumption in network interfaces and routers. Our proposed algorithm performs selection of which buffers to implement and which network interfaces and routers the memories should be attached to minimize the energy consumption.

4. APPROACH

4.1 Architectural Model

The problem of memory and NoC synthesis is the task of mapping arrays (logical buffers) to physical memories, mapping these physical memories along with other cores to routers and determining static routes for the data between cores, while trying to minimize the total energy consumption.

Recollect that in our approach the scratch pad memories which hold circular buffers from the reuse graph are connected to network interfaces as cores (Figure 1). We also make an assumption that all processing cores have their own small private memories, which hold small arrays, local variables and are also used for allocating the stack. NoC (and data reuse buffers described above) are used only for transferring the data shared between several processors or for accessing large arrays of multimedia data (such as frame or image) that cannot fit into small local memories. Our analysis shows that for such separation of data 1-2 KB of local processor memory is enough for most of the typical multimedia benchmarks, such as those described in Section 5. The rest of the memory requests, not handled by the local processor memories, are analyzed by our data reuse analysis technique [10] to obtain a set of possible intermediate data reuse buffers, which can hold the data that are reused many times in additional small memories near the cores that consume that data, thus reducing access time and energy consumption. The use of memories for such buffers instead of caches, in addition to lower power consumption per access, creates a system with fully predictable timing and thus eliminated the need for overdesigning the communication architecture to meet the worst case constraints.

We also assume that the memories we are using are fast enough to provide required bandwidth.

To model off-chip main memory, we represent it as a core whose mapping is fixed to the middle router in the first row of routers in NoC mesh.

We modeled the Æthereal NoC in our experiments. This is a synchronous NoC, which means that all the links and routes are operating on the same NoC frequency. However, network interfaces provide clock conversion, which mean that cores are not required to operate on the same frequency as NoC. Thus, the NoC frequency can be adjusted independently of that of cores. During the NoC synthesis we select its frequency to be the lowest possible that still provides enough bandwidth to perform all the required NoC data transfers in the allotted time.

In the next subsections we present two heuristics, namely the traditional two-step and our co-synthesis heuristic. The first one is the traditional

approach to system design, when memory synthesis is followed by NoC synthesis. The second, our proposed co-synthesis heuristic, performs these two tasks simultaneously. We compare effectiveness of these approaches in Section 5.

Algorithm 1. Two-step synthesis heuristic
input: reuse graph *RG*;
set of nodes *B* that represent memory buffers in the reuse graph;
set of routers RT of a mesh-type NoC
output: synthesized NoC

1. Create min memory energy communication graph CG out of the reuse graph:
1.1. Create communication graph CG out of main memory and processor nodes of the reuse graph
1.2 do {
1.3. for all nodes $b \in B$ do {
1.4. Add b to communication graph CG
1.5. Estimate memory energy
1.6. Remove b from communication graph CG
1.7. }
1.8. Add node $b_{best} \in B$ that reduces energy the most to CG (if found)
1.9. } while such node b_{best} is found

2. Perform NoC synthesis using communication graph CG:
// Initial placement
2.1. N = a set of CG nodes
2.2. Find a node $b \in N$ with the max communication requirements
2.3. Map b to the router in the center of NoC mesh
2.4. while there are unmapped nodes do {
2.5. Find unmapped node $b \in N$ which communicates the most with the mapped nodes
2.6. Map b to a router so that communication cost is minimized
2.7. }
// Mapping improvement
2.8. Perform NoC routing and floorplanning
2.9. E_{best}=estimated energy
2.10. for all $r_1 \in RT$ do {
2.11. if no cores are mapped to r_1, continue
2.12. swap_type=0
2.13. for all $r_2 \in RT$ do {
2.14. perform swap type 1 of routers r_1 and r_2
2.15. perform NoC routing and floorplanning
2.16. E_1=estimated energy
2.17. undo the swap
2.18. if ($E_1 < E_{best}$)
2.19. $E_{best} = E_1$; swap_type=1; $r=r_2$
2.20. perform swap type 2 of routers r_1 and r_2
2.21. perform NoC routing and floorplanning
2.22. E_2=estimated energy
2.23. undo the swap
2.24. if ($E_2 < E_{best}$)
2.25. $E_{best} = E_2$; swap_type=2; $r=r_2$
2.26. }
2.27. if (swap_type = 1)
2.28. perform first type of swap on routers r_1 and r
2.29. if (swap_type = 2)
2.30. perform second type of swap on routers r_1 and r
2.31. }
2.32. perform NoC routing and floorplanning
2.33. E_{best}=estimate_energy()
2.34. return

Figure 4. Two-step synthesis heuristic

4.2 Two-Step Memory/NoC Synthesis Heuristic

The first heuristic consists of two parts. First, we determine which buffers from the reuse graph should be implemented in order to achieve minimum energy consumption in memories, regardless of communication energy. The selected set of buffers together with the traffic between them forms communication graph, which is then used to perform NoC synthesis with the goal of reducing communication energy.

The outline of the heuristic is shown in Figure 4.

In the first part of the heuristic (Step 1), we start from the communication graph that includes only processors and main memory out of the reuse graph nodes obtained by the DRDM technique [10] (Step 1.1). Then we find a buffer (or a set of symmetric private buffers, which have the same size and configuration but belongs to the different processors) that result in the maximum possible decrease of total memory energy when added to the communication graph (Steps 1.3 – 1.7). If such node (or set of nodes) is found, we add it to the communication graph (Step 1.8) and repeat this process again. The memory energy is estimated based on the number of accesses to each buffer (this information is extracted from the reuse graph) and on the memory energy per access.

The second part of the heuristic, the NoC synthesis (Step 2), is based on the NMAP algorithm for single minimum-path routing from [16]. We improved the algorithm to allow mapping of several memories (or processor and memories) to the same router. This approach better exploits the Æthereal architecture in which several cores can be connected to the same router. Our experiments show that this feature often leads to solutions with lower overall energy consumption.

The NoC synthesis consists of two parts. In the beginning, the initial mapping of communication graph nodes to routers is obtained (Steps 2.1 – 2.9). Only one node is allowed per router. Then, in Steps 2.10 – 2.34, nodes are remapped to achieve the lowest possible NoC energy consumption.

During the initial mapping phase, we start with the communication graph node that has the highest communication demands. We map it to the central router in the mesh-type NoC (Steps 2.2 – 2.3). Then, we select a node that communicates the most with already mapped node(s) (Step 2.5), and map it to such a router that the communication cost with the already mapped nodes is minimized (Step 2.6). Communication cost is calculated as a sum of products of bandwidth requirements of the communication graph edge and the minimum distance (in NoC links) between the routers to which the nodes connected with that edge were mapped. The loop (Steps 2.4 – 2.7) is repeated until all nodes are mapped.

In the mapping refinement phase (Steps 2.8 – 2.33), we start with initial estimate of memory and NoC energy consumption (Steps 2.8 – 2.9). For that, routing (mapping of communication graph edges to the NoC links) and floorplanning (needed to estimate NoC link lengths) is done.

Minimum-path routing (Step 2.8) is performed by executing Dijkstra's shortest path algorithm for every communication graph edge to be routed, in the decreasing order of edges bandwidth requirements. Dijkstra's algorithm is performed on a smallest rectangular subset of NoC mesh that includes the routers to which the ends of the edge are mapped to. The bandwidth requirements of the communication graph edges that have been already mapped to NoC links are used as distances for the Dijkstra's algorithm. To approximate NoC link lengths, we use area numbers from memory and processor models and assume that the cores are of square shape.

The total energy consumption is calculated by adding memories, NoC links, routers and NIs energy consumptions (Step 2.9). Routers and NIs have two energy components: one that is consumed every clock cycle, and one that is proportional to traffic. The total number of clock cycles is determined by the busiest NoC link (thus we assume that NoC is running as slow as possible to still meet the deadline).

Next, we try to improve the current mapping by swapping the cores connected to the routers (Steps 2.10 – 2.31). We use two types of swapping: first type is when we completely swap all the NIs and cores between the two routers (Steps 2.14 – 2.19), and the second type is when we try to move memory node (excluding main memory) from the first router to the second (Steps 2.20 – 2.25). We leave the swaps that reduces the total energy the most (Steps 2.27 – 2.30), and undo the rest.

The complexity of the heuristic is $B^3+BR^2+ER^3lgR$, where B and E – the number of nodes and edges in the reuse graph, and R – the number of routers in NoC. If the number of reuse graph nodes and the number of routers are of the same order of magnitude (which is a reasonable assumption), the complexity is ER^3lgR, the same as of the NMAP NoC synthesis algorithm [16].

In the next subsection we present a co-synthesis algorithm, where the memory subsystem is customized together with NoC synthesis.

4.3 Memory/NoC Co-Synthesis Heuristic

The second heuristic is a co-synthesis heuristic, i.e. the memory synthesis is performed simultaneously with NoC synthesis.

The outline of the heuristic is shown in Figure 5.

In the first part of the heuristic (Step 1) we set a goal of reducing overall power consumption by reducing the NoC clock frequency, since as it was

shown in [21], the biggest contributor of additional power consumption after replacing buses with Æthereal NoC in a TV companion chip was the NoC clock power (54%). Due to this reason our heuristic first tries to reduce the maximum bandwidth requirements among all links between routers and between routers and NIs by performing memory customization.

In the second part of the heuristic (Step 2), we try to add additional buffers from the reuse graph to see if they can further reduce memory/NoC power consumption.

Algorithm 2. Co-synthesis heuristic
input: reuse graph *RG*;
set of nodes *B* that represent memory buffers in the reuse graph RG;
set of links (between routers and between routers and NIs) L of the NoC
output: synthesized NoC

1. Try to add buffers so that NoC clock frequency is reduced:
1.1. Create communication graph CG using main memory and processor nodes from the reuse tree
1.2. Synthesize(CG) // synthesize NoC as in step 2 of Alg. 1
1.3. do {
1.4. Find a link l∈L with the highest bandwidth
1.5. for all comm. graph edges cgl mapped to link l, in the order of decreasing bandwidth, do {
1.6. M = set of reuse graph buffers that affect edge cgl
1.7. Among all buffers b∈M, find the one b_{best} that reduces total energy the most
1.8. If found, add b_{best} to CG; go to step 1.11
1.9. }
1.10. go to step 2.1
1.11.} while (true)

2. Try to add other memory buffers to further reduce total energy consumption:
2.1. E_{best}=estimated energy
2.2. do {
2.3. Find b∈B that have not been tried to be added to CG before and with the highest reduction of traffic
2.4. Add b to CG
2.5. Synthesize(CG)
2.6. Remove b from CG
2.7. E=estimated energy
2.8. if (E< E_{best})
2.9. E_{best} = E; Add b to CG
2.10. }
2.11. return

Figure 5. Co-synthesis heuristic

In the first part of the Algorithm 2, we create communication graph CG using main memory and processor nodes from the reuse graph (Step 1.1). In Step 1.2 we synthesize NoC using the communication graph CG as it was explained in Step 2 of the Algorithm 1. In Step 1.4 we find the NoC link *l* with the highest bandwidth requirements. In Step 1.5, we analyze the communication graph edges that were routed through NoC link *l* and iterate over edges in the order of decreasing contribution to the NoC link traffic. In Step 1.6 we obtain the set of reuse graph buffers *M*, adding which to the communication graph splits the current edge *cgl* into several edges (with the

added buffers in-between) with different (lower or equal) bandwidth requirements. Next, in Step 1.7, we try to figure out adding which of the buffers (or a set of symmetrical buffers) from M reduces the total energy consumption the most. This step also involves NoC synthesis, which is done using the algorithm from Step 2 of the Algorithm 1. If the buffer that reduces the energy is found, it is added to the current communication graph (Step 1.8) and the loop (Step 1.5) which analyzes and tries to reduce the traffic of the busiest NoC link is repeated. If there are no more buffers that can reduce the maximum NoC clock frequency, the execution proceeds to the second part of the Algorithm 2.

In the second part of the Algorithm 2, we try to find a reuse graph buffer that has not been tried to be added to the communication graph earlier and which reduces the traffic to the higher level of memory hierarchy the most (Step 2.3). We add such buffer to the communication graph and evaluate if this addition reduces total energy consumption (Steps 2.4 – 2.8). If it does, we leave the buffer in the communication graph (Step 2.9) and proceed with the evaluation of other buffers.

The complexity of the heuristic is $B^2R^2+EBR^3lgR$, where B and E – the number of nodes and edges in the reuse graph, R – the number of routers in NoC. If the number of reuse graph nodes and the number of routers are of the same order of magnitude, the complexity is ER^4lgR, which is higher than that of the two-step heuristic.

In the next section we compare and establish the efficacy of the results obtained by the co-synthesis heuristic over the two-step heuristic.

Table 1. Experiment configurations

Experiment number	Benchmark	Processors	Main memory	NoC config
1	QSDPCM	6	800K on-chip	mesh 5x5
2	QSDPCM	6	8MB off-chip	mesh 5x5
3	Laplace	4	800K on-chip	mesh 5x5
4	Laplace	4	8MB off-chip	mesh 5x5
5	Laplace	16	4M on-chip	mesh 6x6
6	Laplace	16	8MB off-chip	mesh 6x6
7	Susan	4	800K on-chip	mesh 4x4
8	Susan	4	8MB off-chip	mesh 4x4

5. EXPERIMENTS

5.1 Experimental Setup

In our experiments we used several typical streaming video and image processing applications: QSDPCM, a video encoder [22], Laplace, image

filtering algorithm, and Susan, image recognition application. All the benchmarks were parallelized and distributed over several processors (Table 1).

Our energy model consists of several components. For on-chip memory power consumption we used (and extrapolated in some cases) the data from MPARM simulator [15]. We used CACTI [18] for 130 nm memory area estimations. We estimated area of TI C64 processors using data from [1][23]. Off-chip memory consumption was calculated based on 8MB Micron low-power CellularRAM chip [14].

Our model for Æthereal NoC power estimation is based on measurements described in [4]. The models for different NoC components are shown in Table 2. Since the network interface energy was not reported, we assumed the energy per port to be similar to that of the router. The area numbers are taken from [3].

Table 2. NoC energy and area

NoC Component	Energy, pJ	Area, mm^2
Router (guaranteed throughput)	E=(16.1+40.3α)*B + 32*P*C α=0.5-activity factor B-number of flits transferred P-number of router ports C-number of NoC clock cycles	0.17 (for 130 nm technology)
Network Interface	same as router	0.13
Link wires	E=(0.27+0.58*L)*N L-length of wire, mm N=32-number of wires	-

5.2 Experimental Results

The list of the benchmarks and parameters we used to synthesize NoC for each of our experiments is shown in Table 1. For each of the benchmarks we synthesized NoC for two cases: assuming that main memory is located on-chip and off-chip. For the latter case, we modified the synthesis algorithms to fix the position of the router to which the main memory was connected to a router on the edge of the chip, as described earlier.

For each benchmark configuration in Table 1, we performed three experiments. First, we synthesized NoC without performing data reuse analysis, i.e., we pruned the reuse graphs to include only main memory and processors. Note that this is the case when both of our heuristics return the same results as NMAP algorithm [16] since no memory synthesis could be performed (the memory subsystem is fixed) and processors and main memory are all mapped to different routers (due to restrictions in our modified NMAP synthesis algorithm) as it happens in the original NMAP

algorithm. Second, we synthesized NoC using sequential two-step synthesis heuristic (Algorithm 1 in Section 4). Finally, we performed memory/NoC co-synthesis using Algorithm 2 in Section 4.

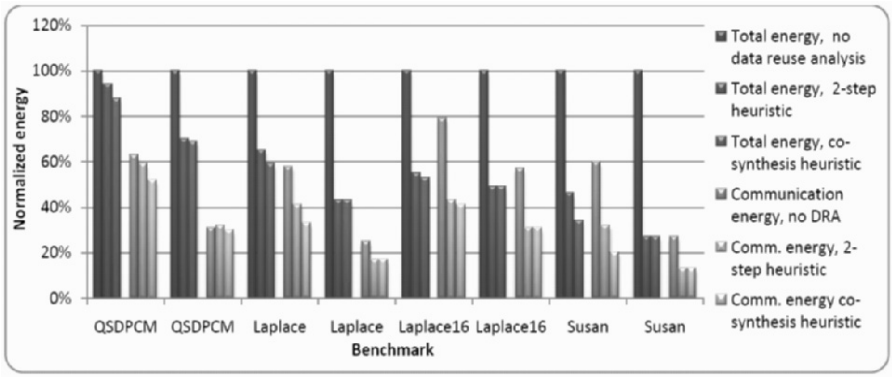


Figure 6. Experimental results

The results are shown in Figure 6. The graph shows energy consumption of the memory and communication subsystems normalized to the first case, when data reuse analysis is not used. For each benchmark, the first three graphs show the combined energy consumption of memory and NoC, and the last three graphs show the energy consumption of NoC alone. Results shows that customizing memory subsystem by employing data reuse analysis allows to reduce communication energy by 31% on average, and total energy by 44% on average. Furthermore, the use of memory/NoC co-synthesis heuristic instead of two-step synthesis allows to further reduce the communication energy by up to 38% (10% on average) and total (memory and communication) energy by up to 26% (6% on average).

6. CONCLUSION

In this work we investigated the influence of memory subsystem on power consumption of NoC communication architecture. We showed that customizing memory subsystem by employing data reuse analysis allows reducing communication energy by 31% on average. We also proposed a data reuse based memory/NoC co-synthesis heuristic, and showed that it outperforms traditional two-step synthesis approach by reducing the communication energy by up to 38% and total energy by up to 26%. Although our approach was experimented on the Æthereal NoC architecture, we believe this co-synthesis strategy will be useful for a wide range of other NoC architectures. Future work will investigate these issues.

REFERENCES

[1] S. Agarwala et al. A 600-MHz VLIW DSP. IEEE Journal of Solid-State Circuits, Nov. 2002.

[2] W. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In Proc. of DAC, June 2001.

[3] J. Dielissen et al. Concepts and Implementation of the Philips Network-on-Chip. IP-Based SOC Design, Grenoble, France, Nov. 2003.

[4] J. Dielissen et al. Power Measurements and Analysis of a Network on Chip. Philips Research, Technical Note TN-2005-0282, Apr. 2005.

[5] K. Goossens et al. Æthereal Network on Chip: Concepts, Architectures, and Implementations. IEEE Design and Test of Computers, 2005.

[6] P. Guerrier and A. Greiner. A generic architecture for on-chip packet switched interconnections. In Proc. of DATE, March. 2000.

[7] Zvika Guz et al. Efficient Link Capacity and QoS Design for Network-on-Chip. In Proc. of DATE, March 2006.

[8] R. Ho, K. Mai, and M. Horowitz. The future of wires. In Proc. of the IEEE, April. 2001.

[9] Hu et al. Energy-Aware Mapping for Tile-based NoC Architectures Under Performance Constraints. In Proc. of ASP-DAC, 2003.

[10] I. Issenin, E. Brockmeyer, B. Durinck, N. Dutt. Multiprocessor System-on-Chip Data Reuse Analysis for Exploring Customized Memory Hierarchies. In Proc. of DAC, 2006.

[11] I. Issenin, N. Dutt. Data Reuse Driven Energy-Aware MPSoC Co-Synthesis of Memory and Communication Architecture for Streaming Applications. In Proc. of CODES+ISSS, 2006.

[12] S. Kim et al. Efficient Exploration of On-Chip Bus Architectures and Memory Allocation. In Proc. of CODES+ISSS, 2004.

[13] Manolache et al. Fault and Energy-Aware Communication Mapping with Guaranteed Latency for Applications Implemented on NoC. In Proc. of DAC, 2005.

[14] Micron Async/Page/Burst CellularRAM Memory MT45W4MW16BCGB, http://download.micron.com/pdf/datasheets/psram/64Mb_burst_cr1_5_p25z.pdf.

[15] MPARM project, http://www-micrel.deis.unibo.it/sitonew/ research/mparm.html.

[16] S. Murali, G. De Micheli. Bandwidth-Constrained Mapping of Cores onto NoC Architectures. In Proc. of DATE, 2004.

[17] S. Pasricha, N. Dutt. COSMECA: Application Specific Co-Synthesis of Memory and Communication Architectures for MPSoC. In Proc. of DATE, 2006.

[18] P. Shivakumar, N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Technical Report 2001/2, Aug. 2001.

[19] K. Srinivasan et al. A Technique for Low Energy Mapping and Routing in Network-on-Chip Architectures. In Proc. of ISLPED, 2005.

[20] K. Srinivasan et al. Layout Aware Design of Mesh based NoC Architectures. In Proc. of CODES+ISSS, 2006.

[21] F. Steenhof et al. Networks on chips for high-end consumer-electronics TV system architectures. In Proc. of DATE, 2006.

[22] P. Stobach. A new technique in scene adaptive coding. In Proc. of EUSIPCO, Grenoble, 1988.

[23] Z. Yu et al. An Asynchronous Array of Simple Processors for DSP Applications. In Proc. of ISSCC, 2006.

EFFICIENT AND EXTENSIBLE TRANSACTION LEVEL MODELING BASED ON AN OBJECT ORIENTED MODEL OF BUS TRANSACTIONS

Rauf Salimi Khaligh, Martin Radetzki
Institut für Technische Informatik,Universität Stuttgart
Pfaffenwaldring 47
D-70569 Stuttgart, Germany
rauf.salimi@informatik.uni-stuttgart.de
martin.radetzki@informatik.uni-stuttgart.de

Abstract Transaction level modeling (TLM) aims at abstract description and high-performance simulation of bus-based System-on-Chip (SoC) platforms. While languages and libraries providing the necessary basic modeling mechanisms are available, there is ongoing discussion on modeling styles for their usage. This contribution employs object-oriented modeling of transaction data and bus protocols, targeting at better extensibility and reuse of models. Our simulation performance measurements show that by abstracting from signal-level data representations, a cycle-accurate object-oriented model can achieve performance close to models on cycle-approximate level.

Keywords: Transaction-Level Modeling, SystemC, Object-Oriented, Embedded Systems

1. INTRODUCTION

Embedded systems and systems on chip integrate an increasing number of processor nodes and complex communication structures. It is an essential design task to model these systems prior to their implementation. This enables an early validation of system architecture concepts, exploration of architectural alternatives, and performance evaluation. Modeling on register transfer level (RTL) is no longer an option due to the complexity, inflexibility, and low simulation performance of the resulting models. Transaction level modeling has been devised as an alternative.

One key principle of TLM is to perform communication by function calls rather than via signal communication. The other is to exchange information on a coarser level of granularity compared to low-level signals. With these constraints, bus based systems can be described at different TLM abstraction

Khaligh, R.S., Radetzki, M, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 313–324.

levels. The abstraction levels representing different levels of accuracy required in most modeling activities have been identified and proposed by some researchers and institutes active in the transaction level modeling field (e.g. [6, 5, 12]). For example, in The most abstract level, the so-called functional view (FV) abstracts completely from the communication architecture by modeling point-to-point connections between each pair of communication partners. In the programmer's view (PV), bus communication is modelled under abstraction from timing and arbitration. The architecture view (AV), also called cycle approximate (CX) view, adds approximated arbitration and timing. In the cycle accurate (CA) or verification view (VV), transactions must allow to reproduce for each cycle the bus owner and transaction that is active in an RT level implementation of the bus.

The SystemC TLM standard defines interfaces for initiating and receiving transactions as well as basic TLM fifo channels. It does not describe a systematic modeling style for the transactions themselves. The contribution of this work is an object-oriented modeling of transactions that leads to easily extensible cycle-accurate models which approach the simulation speed of less precise CX models.

The remainder of this paper is organized as follows. In Section 2, we summarize related work. Section 3 provides a summary of our object-oriented transaction models. In Section 4, we present its application to cycle-accurate modeling of the AMBA AHB. Section 5 presents experimental results on the achievable simulation performance, and Section 6 concludes this paper.

2. RELATED WORK

The essence of transaction level modeling (TLM) is language and application domain independent. Examples of modeling languages explicitly enabling transaction level modeling are SystemC [8] and SpecC [7]. In the relatively young field of TLM, communication schemes, interoperability, definition of abstraction levels and approximations and the terminology are constant subjects of debate and standardization effort. Examples are works of the Open SystemC Initiative (OSCI), the Open Core Protocol International Partnership (OCP-IP) and the SpecC modeling community [3, 6, 5, 12].

In its current version (1.0), the OSCI-TLM standard [12] proposes use of certain interfaces and channels based on these interfaces. The OSCI-TLM standard addresses several issues such as separation of user code from the communication code in layers, module interoperability and reuse. However, the modeling style presented in the OSCI-TLM white paper has a number of drawbacks. Modeling passive components deviates significantly from modeling active components based on the presented modeling style. The transition from passive to active models is often required when moving from higher to

lower levels of abstraction. Another drawback is use of C *structs* to transfer information. This affects the extensibility and reuse in the models. These issues have been addressed in [11] and a new modeling style – the Object Oriented Transaction Level Modeling (OOTLM) approach – was proposed and demonstrated using a simple model. However, the applicability of this method to complex real-world communication protocols was not proven and its performance implications and accuracy limitations were not studied in detail. In this paper a protocol widely used in embedded systems, the AMBA AHB [1] is modelled based on the OOTLM approach to investigate its applicability and the achievable simulation performance.

A recent work in the OSCI-TLM domain is GreenBus [9] which provides a generic, protocol neutral interconnection scheme and library for SystemC models on top of the OSCI-TLM standard which can be customized to model concrete bus protocols. In GreenBus, a bus transaction is composed of uninterruptible transfers called *atoms* which transfer basic data elements referred to as *quarks*. Granularity of the information being monitored or updated by components connected to a GreenBus based bus depends on the abstraction level of the model.

The AMBA AHB has been used by many researchers in evaluation of their modeling approaches. In [10] Pasricha et al have developed models of the AHB based on their proposed CCATB abstraction level. In [13] Schirner et al compare accuracy and simulation performance of their SpecC based AHB models in different levels of abstraction. Caldari et al [4] have developed transaction level models of the AHB in SystemC 2.0. Their work is not based on the OSCI-TLM standard. Recently the ARM specific Cycle Accurate Simulation Interface (CASI) and CASI based AMBA models have been released by ARM [2]. CASI models are essentially cycle-based models, and to achieve high simulation speeds avoid using SystemC events (except for clock events).

3. THE OOTLM APPROACH

In OOTLM, the interactions between model elements are represented by transaction objects. A transaction object represents read or write transfers, reset or initialize operations or abstractions of more complex interactions and encodes the information required for the particular interaction. For example, for a reset operation, this would be the address of the target to be reset. For a read transfer, attributes of the transaction object represent the address of the data item to be transfered and the data to be returned by the target.

Transaction objects are created and initialized by the initiating masters and are sent to the bus model via OSCI-TLM compliant channels. Subject to the arbitration policy and algorithm, the transaction objects are eventually routed to the addressed slaves to be *processed*. For example, for read transfers, the

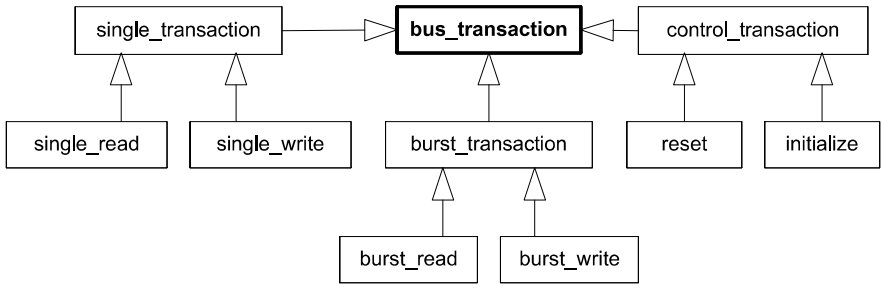


Figure 1. A generic transaction class hierarchy

slave updates the transaction object with the correct response and data values. During the life time of a transaction object, the initiating master has access to it through a pointer and can query or change its attributes. This way, without loss of precision, the additional data traffic and the overhead of using separate request, response and control transactions is avoided. A generic example of transaction classes that could be used to model the operations in a typical bus is shown in the UML class diagram of figure 1. Using an appropriately designed transaction class hierarchy, a single OSCI-TLM based channel can be used to transport different types of transaction objects (e.g. with `put(T &t)` and `get(T &t)` methods). In this example this would be a channel of type `tlm_fifo<bus_transaction*>` and would result in the simple interconnection scheme shown in figure 2.

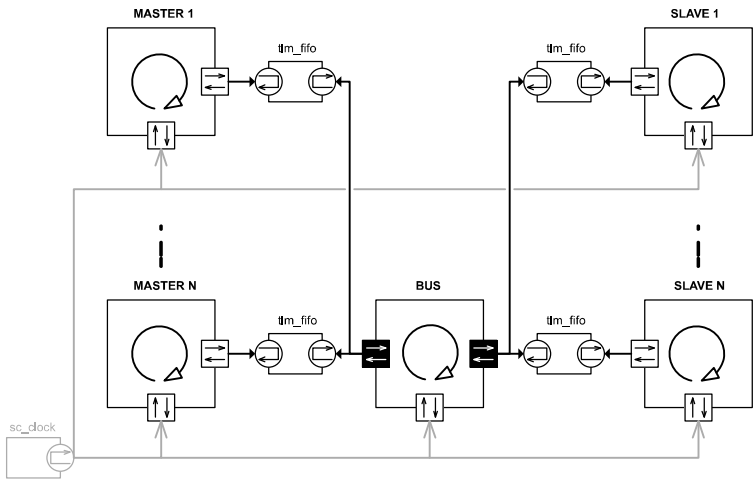


Figure 2. Model interconnection

As a modeling technique enabling reuse of passive slave models (see [11]), interaction of the slaves with the transaction objects is performed using the

existing methods of the slaves (e.g. `read()` and `write()`). The next sections also show how the OOTLM approach enables reuse, incremental development and extensibility of the models.

4. A CYCLE ACCURATE OOTLM MODEL OF THE AMBA AHB

The Advanced High Performance Bus (AHB) is part of the AMBA family of busses from ARM [1] and is intended to be used as a backbone bus in systems-on-a-chip (SoCs). The AHB is a multi-master, pipelined bus featuring single cycle bus handover, fixed size bursts, incremental bursts of unspecified length and split and retry transfers.

In this work, we have focused on a subset of the AHB specification large enough to validate the OOTLM approach in modeling complex bus protocols. We have modelled single data item transfers and fixed-length bursts. From the possible slave responses, we have chosen `OKAY`, `ERROR` and `SPLIT` responses. As a simplifying assumption, preemption of unlocked bursts has not been modelled and a burst can only be terminated prior to its completion as a result of a `SPLIT` response. However as mentioned in section 4.1, this does not affect the generality of our approach and the developed models can be modified to support unlocked bursts in a similar fashion to split transfers.

The developed model is cycle accurate. The arbitration, bus handover, inter and intra-transaction timing and status of the transfers in each cycle are accurate and fully compliant with the AHB specification.

The basis of our model is the pipelined nature of the AHB, shown in an abstract form in figure 3. AHB transfers have two phases, the *address phase* and the *data phase*. In each bus cycle, two different transfers may be underway concurrently on the bus. One in the address phase and one in the data phase. A transfer in the address phase progresses to the data phase, when the transfer currently in the data phase completes.

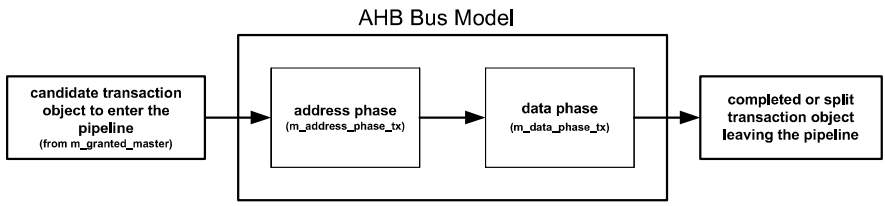


Figure 3. An abstract view of the AHB bus as a pipeline

Within the bus model (section 4.2), this abstract pipelined view of the AHB is maintained and updated in each cycle using two pointers, one to the transaction in the address phase (`m_address_phase_tx`) and one to the transaction in the

data phase (`m_data_phase_tx`). Depending on status of `m_data_phase_tx`, the bus model coordinates routing of the transaction objects and other timing aspects of the AHB bus.

4.1 AHB Data Transfer Transactions

The transaction objects are the means of information transfer between masters and slaves and can be considered virtual communication paths created dynamically by the bus model. Figure 4 shows the transaction class hierarchy modeling the AHB single transfers and bursts. Some attributes represent the address, data and control information and have direct counterparts in the AHB signal set (e.g. `m_transfer`). Other attributes are used to model the dynamics of the transaction objects, delayed data bus handover and split transfers. The slave response and ready status are visible to the initiating master and can be polled. Alternatively, a master can wait for the completion of transaction without polling by waiting on `m_completion_event`.

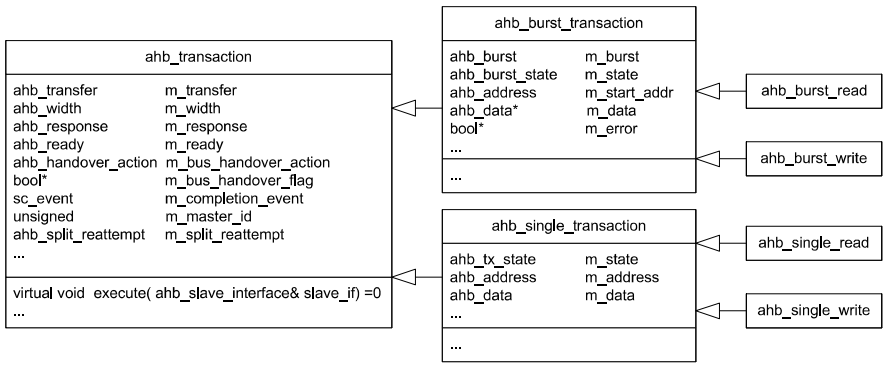


Figure 4. AHB transaction class hierarchy

The slave processing a transaction object calls the `execute()` method, passing a pointer to its `ahb_slave_interface` (figure 6), to update the attributes of the transaction object. To get the data, response and ready status, the `execute()` method will in turn call the `read()` and `write()` interface methods of the target slave whenever necessary.

Based on the properties of the bursts and split transfers, the amount of object traffic between model elements is reduced to improve performance. For split transfers, the bus and slave models keep pointers to the split transaction objects in a list of *pending* transactions. The slave signals to the arbitration algorithm its readiness to process the transaction using the `m_split_reattempt` attribute of the transaction object. Similarly, the re-attempting master does not need to send a new transaction object to the slave, the bus model re-enables the transaction object to be re-processed by the slave. AHB bursts are not allowed

to cross slave address boundaries. Thus, for every burst a single transaction object is sent to the addressed slave.

Although not modeled and implemented in this work, preemption of unlocked bursts (as the result of the master loosing the bus) can also be modeled similarly with the aforementioned *pending* transactions, with simple modifications to the model. The initiating master of a burst can not perform any other transactions until all beats of the burst are completed either successfully or with an error. Again, a single burst transaction object needs to be sent to the slave by the master. If the master loses the bus prematurely, the burst transaction will become a pending transaction, waiting to be re-enabled by the bus model when the master takes ownership of the bus.

The dynamic behavior of the transaction objects is modelled by state machines. The state machine for `ahb_single_transaction` is shown in figure 5. Burst transactions are modelled with a concurrent state machine, observing the fact that a burst can have concurrent beats, one in the address phase and one in the data phase. The burst state machine is omitted here to save space.

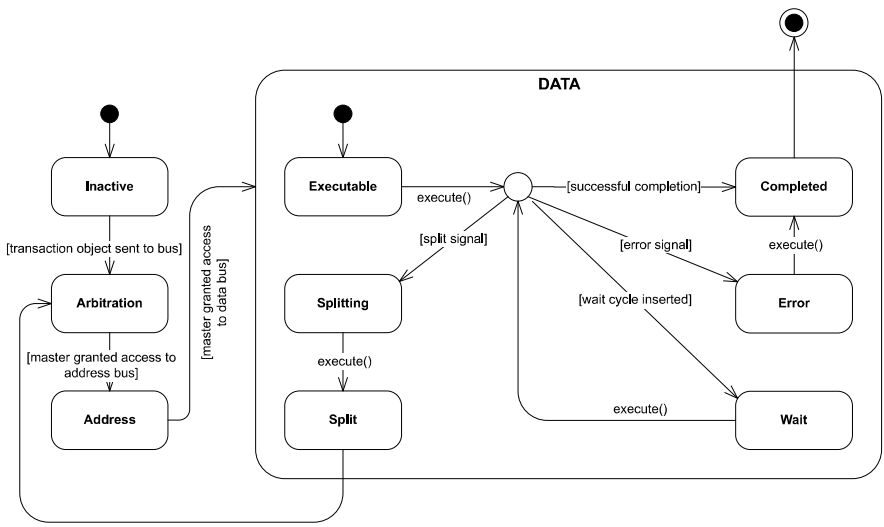


Figure 5. Single transaction state machine

4.2 The AHB Bus Model

The bus model is responsible for routing of the transaction objects to their addressed slaves and is partially shown in figure 6. The arbitration algorithm is implemented in the `select_master()` method and considers transactions in the master-bus channels and the pending transaction list in selecting the highest priority master. An entry in the pending transaction list corresponds to the

request of a master which has received a split transfer and is waiting to be granted the bus for a re-attempt. A data structure mapping address ranges to slaves is used for decoding, and the transaction objects are routed to the addressed slaves either via the slave-bus channel (for new transactions) or by putting the transaction objects in the address phase again (for split transactions, figure 5).

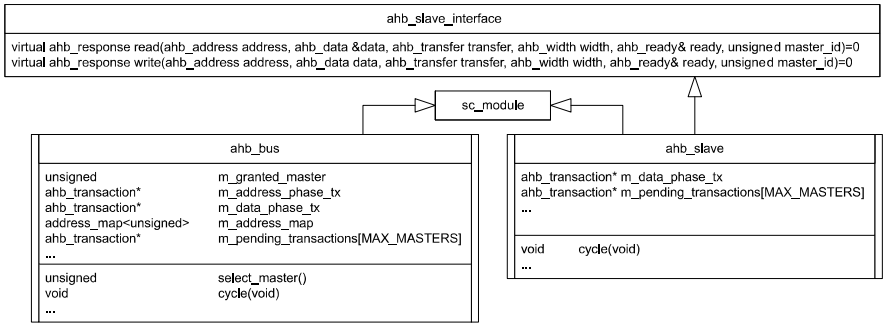


Figure 6. Bus and slave models

The functionality of the bus is implemented in the clock triggered process `cycle()` and is modelled by a state machine (figure 7). Based on the pipeline model of the bus (figure 3), arbitration and all aspects of the AHB bus are modelled accurately using this state machine.

Together with the bus model, we also developed generic master and slave models for performance measurement and validation purposes. Because of the inheritance relationship between the transaction classes and since the models were designed based on a set of polymorphic methods of the transaction classes, we were able to develop the models incrementally and in three steps, extending the models by subclassing. In the first step, the basic elements of the AHB functionality – arbitration, decoding/routing, pipelined behavior and single transactions – were modelled and implemented. In the second step split transactions, and in the last step bursts and burst related issues were implemented. For example, to extend the bus model from the first step to support split transactions, one new transition was required in the bus state machine, and the model was extended by adding attributes, one new event and overriding some methods. As another example, no changes were necessary in the slave from the second step to enable processing of burst transactions.

5. EXPERIMENTAL RESULTS

We implemented the models using the OSCI SystemC library version 2.1 v1 with Microsoft Visual C++ .NET 2003 in Microsoft Windows XP professional. The bus model was validated against the protocol specification using

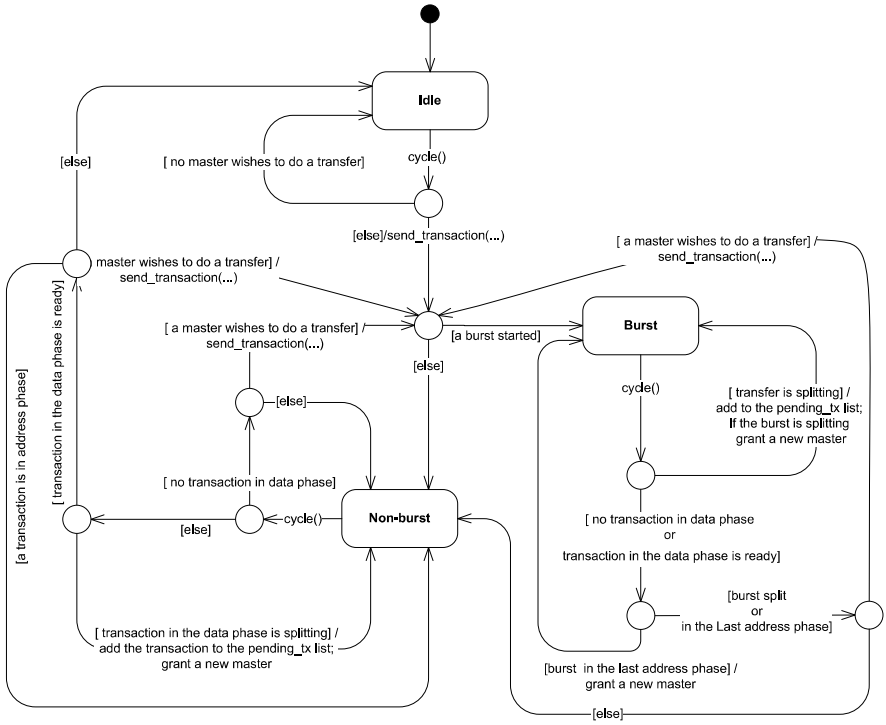


Figure 7. Bus state machine

deterministic and randomized tests. To measure the simulation performance of the bus model, we used a single master, single slave test setup similar to [13] and measured performance indices also reported in other related work [9, 4].

First, we measured the simulated transaction processing time and equivalently the simulated bandwidth. These indices represent the minimum amount of (real or wall clock) time required to perform a data transfer transaction of a certain size using the most efficient combination of bursts and single transfers. Figure 8 shows the measured simulated bandwidth for transactions ranging in size from 1 byte to 1024 bytes. Each transaction was performed 1000 times (all with `OKAY` responses and no wait cycles), and the average time was recorded. The measurements were performed on a workstation with a 2.21 GHz AMD Athlon(tm) 64 3500+ processor with 512 kb of Cache and 1 GB of RAM.

The saw-tooth shape of the graph is the result of breaking the transfers into a combination of bursts and single transfers by the master. For small transfers, the overhead of transfer of transaction objects is relatively high and the worst case bandwidth reaches a value of 0.270 MBytes/Sec. As the transfer size increases, this overhead becomes negligible and the bandwidth approaches a value of 1.2 MBytes/Sec.

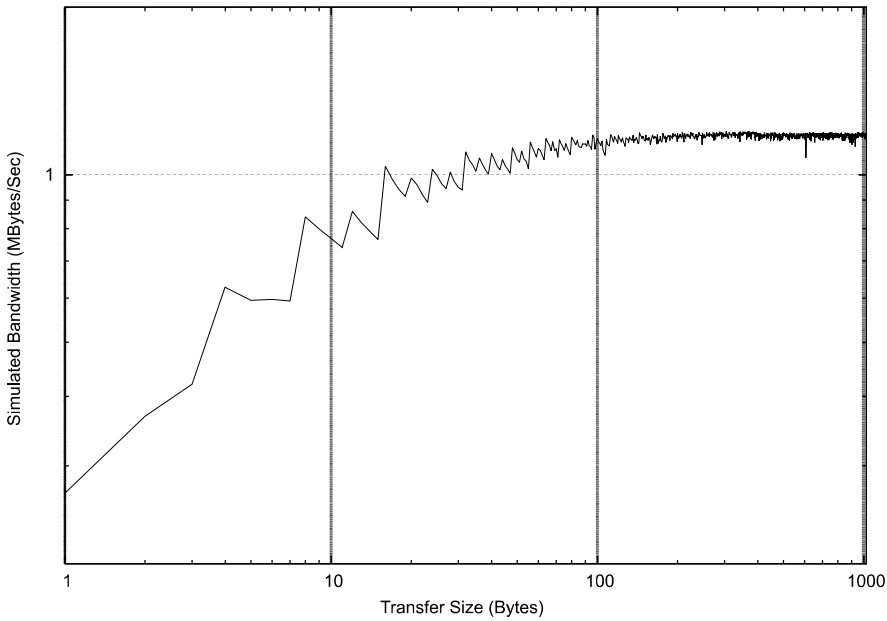


Figure 8. Simulated bandwidth

In a similar test setup but in a different simulation environment, the bus functional model (BFM) of the SpecC-based model of Schirner and Dömer [13] reaches a simulated bandwidth of 0.03 MBytes/Sec and their next most accurate model (ATLM, which is essentially a cycle-approximate model), reaches 2.29 MBytes/Sec. In [9] performance figures of bus accurate GreenBus based models of the IBM PLB are presented, which expressed in terms of simulated bandwidth, reach a maximum of 2.8 MBytes/Sec.

It should be noted that our reported simulated bandwidth values are very conservative. We have used a single data width for all transfers (e.g. byte). For example, in our test setup, to transfer three bytes, three bus transactions are used, each transferring a single byte. In the ATLM model in [13] on the other hand, three bytes are transferred in two transactions, one transferring a byte and one transferring an AHB half-word (two bytes). In our model, transferring three individual words (each word being four bytes) would take the same amount of time required to transfer three individual bytes. Considering this, the simulated bandwidth of our model approaches 4.8 MBytes/Sec. However, we have decided to report the more conservative values, as the aforementioned details regarding transfer setup were not explained in some published related work (e.g. [9]).

Next, we measured the model simulation speed which is expressed in terms of simulated number of clock cycles of a simulation model per unit of time.

The average model simulation speed for our test setup reached 1150 KCycles/Sec. Caldari et al [4] have reported a simulation speed of 300 KCycles/Sec for their SystemC 2.0 based models. They have used a more complex master model and have performed their measurements on a Sun Ultra 60 Workstation.

6. CONCLUSIONS

We have shown that, based on an abstract object-oriented modeling style, easily extensible and cycle accurate modeling of a complex, real world bus system and protocol can be performed. Instead of using discrete transactions for arbitration, control, address and data, we have modelled bus transactions using complex transaction objects which encode accurately all the necessary information, and whose dynamic behavior represents different phases of the bus transfers accurately down to the individual buy cycles. This results in reduced traffic between model elements which in turn leads to higher simulation performance, without loss of precision. Our measurements, in comparison to the related work, show a simulation performance that is significantly above other cycle-accurate models and comes close to the performance of models that are cycle-approximate. By further reducing the amount of SystemC events and by moving to the cycle-approximate level, we plan to achieve a further speed-up. Future work includes the application to additional bus protocols and the development of a simulation library of master and slave components as well as a connection of our model to instruction set simulators.

References

[1] ARM Limited. *AMBA Specification, version 2.0*, May 1999.

[2] ARM Limited. *AMBA AHB transaction level modeling specification, version 1.0.1*, May 2006.

[3] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of CODES+ISSS '03*. ACM Press, 2003.

[4] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. Transaction-level models for AMBA bus architecture using SystemC 2.0. In *Proceeding of DATE '03*. IEEE Computer Society, 2003.

[5] J. A. Colgan and P. Hardee. *Advancing Transaction Level Modeling, Whitepaper*. CoWare, Inc., 2004.

[6] A. Donlin. Transaction level modeling: flows and use models. In *Proceedings of CODES+ISSS '04*. ACM Press, 2004.

[7] R. Dömer, A. Gerstlauer, and D. Gajski. *The SpecC Language Reference Manual, version 2.0*. SpecC Technology Open Consortium, Dec 2002.

[8] IEEE Computer Society. *IEEE Standard SystemC language reference manual,IEEE Standard 1666-2005*, 2005.

[9] W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, and M. Burton. Greenbus: a generic interconnect fabric for transaction level modelling. In *Proceedings of DAC '06*. ACM Press, 2006.

[10] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Fast exploration of bus-based on-chip communication architectures. In *Proceedings of CODES+ISSS '04*. ACM Press, 2004.

[11] M. Radetzki. SystemC TLM Transaction Modelling and Dispatch for Active Objects. In *Proceedings of FDL'06*, Darmstadt, Germany, 2006.

[12] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. *Transaction level modeling in SystemC, Whitepaper*. Open SystemC Initiative, 2004.

[13] G. Schirner and R. Dömer. Quantitative analysis of transaction level models for the AMBA bus. In *Proceedings of DATE '06*. European Design and Automation Association, 2006.

HARDWARE IMPLEMENTATION OF THE TIME-TRIGGERED ETHERNET CONTROLLER*

Klaus Steinhammer and Astrit Ademaj
Vienna University of Technology, Real-Time Systems Group
Treitlstr. 3/182-1, A-1040 Vienna, Austria
[klaus,ademaj]@vmars.tuwien.ac.at

Abstract: Time-triggered (TT) Ethernet is a novel communication system that integrates real-time and non-real-time traffic into a single communication architecture. A TT Ethernet system consists od a set of nodes interconnected by a specific switch called TT Ethernet switch. A node consist of a TT Ethernet communication controller that executes the TT Ethernet protocol and a host computer that executes the user application. The protocol distinguishes between event-triggered (ET) and time-triggered (TT) Ethernet traffic. Time-triggered traffic is scheduled and transmitted with a predictable transmission delay, whereas event-triggered traffic is transmitted on a best-effort basis. The event-triggered traffic in TT Ethernet is handled in conformance with the existing Ethernet standards of the IEEE.

This paper presents the design of the TT Ethernet communication controller optimized to be implemented in hardware. The paper describes a prototypical implementation using a custom built hardware platform and presents the results of evaluation experiments.

1. INTRODUCTION

The temporal accuracy interval of information in a distributed real-time system is affected by the precision of the global view of time. This precision of the distributed time-base depends on the jitter of the message transmission delay over the network and the communication protocol. A predictable real-time communication system must therefore guarantee the message transmission within

*Acknowledgments: This work has been supported in part by the European IST project DECOS under project No. IST-511764.

Steinhammer, K., Ademaj, A, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 325–338.

a constant transmission delay and bounded jitter [1]. Due to the open nature of Ethernet it is difficult to guarantee bounded transmission delays in systems that use standard Ethernet networks.

The duration of the message transmission over a network depends on the characteristics of the network traffic. It can be distinguished between two different scenarios: *cooperative senders* and *competing senders*. If senders are competing (as in standard Ethernet [2]) there is always the possibility that two or more messages are sent to a single receiver simultaneously. There are two possibilities to resolve this conflict: back-pressure flow control to the sender (this is the choice of the bus-based "Vintage Ethernet" [3]) or the storage of the messages within the network (this is the choice of switched Ethernet). Both alternatives involve increased message transmission jitter which is unsatisfactory from the point of view of real-time performance. It is therefore necessary that in real-time systems the senders must cooperate to avoid message transmission conflicts. There are many approaches that try to adapt the Ethernet technology such that it can be deployed in applications where temporal guarantees are necessary [4–8]. A summary of these implementations can be found on the Real-Time Ethernet web-page [9] while a comparison can be found in [6] and [7]. All these solutions use the concept of *cooperative senders* in order to guarantee constant transmission delays.

This paper presents the design of a Time-Triggered Ethernet Communication Controller (TTE controller) VHDL module which can be applied in an *System On a Programmable Chip* (SOPC) system and a prototypical implementation of a TTE controller using a custom made hardware board.

This paper is organized as follows: Section 2 gives an overview of the TT Ethernet architecture. The design of the TTE Controller is presented in Section 3 while the prototypical implementation is presented in Section 4. The experimental validation results are presented in Section 5, and the paper is concluded in Section 6.

2. TIME-TRIGGERED ETHERNET

A TT Ethernet system consists of a set of nodes interconnected by a specific switch called TT Ethernet switch. A node can be either a *standard Ethernet node* or a *TT Ethernet node*. A standard Ethernet node consist of a "commercial of-the-shelf" (COTS) Ethernet controller and a host computer transmitting only standard Ethernet messages. A TT Ethernet node consist of a TT Ethernet communication controller that executes the TT Ethernet protocol and a host computer that executes the one or more threads of a distributed (real-time) application.

Time-Triggered Ethernet [10] allows competing senders (standard Ethernet node) to *coexist* with cooperative senders (TT Ethernet node) on the same network while yet preserving the temporal predictability of the traffic among the cooperative senders. To integrate competing traffic seamlessly in the same network without interference of the cooperative traffic, Time-Triggered Ethernet introduces two message classes using the standardized Ethernet frame format [2]:

- **Standard Ethernet messages** are used for Event-Triggered (ET) traffic by competing transmitters, and
- **Time-Triggered (TT) messages**, which are transferred among cooperative senders.

Standard Ethernet messages are transmitted in those time intervals where the communication medium is not needed for transmission of time-triggered messages. In order to avoid that standard Ethernet traffic affects the temporal properties of the time-triggered traffic, the TT Ethernet switch preempt all standard Ethernet messages which are in the transmission path of time-triggered messages. This preemption mechanisms allows the transmission of TT messages with a constant transmission delay. As all ET messages are stored within the TT Ethernet switch, the preempted ET messages are retransmitted autonomously by the TT Ethernet switch after the transmission of the TT message has finished. If an ET message has to be dropped by the TT Ethernet switch because of buffer memory overflow, the lost ET message is retransmitted by the upper protocol layers of the sending node. TT messages are not stored within the network.

Introducing these two classes guarantees the compatibility to standard Ethernet COTS components and existing networking protocols without any modification of the competing transmitters on the one hand and the possibility to realize a scheduled message transfer between all cooperative senders by establishing and maintaining a global time base on the other hand.

There are two variations of TT Ethernet: standard TT Ethernet and safety-critical TT Ethernet. Safety-critical TT Ethernet is based on standard TT Ethernet which is described above, but introduces some enhancements to ensure that an arbitrary failure of one device within the cluster is tolerated [11]. This makes TT Ethernet suitable for various application cases - for multimedia applications, the industrial automation up to safety-critical applications like x-by-wire systems in the automotive and aeronautic domain.

The TT Ethernet communication controller presented in this paper is intended to be used in a standard TT Ethernet system.

3. DESIGN OF THE TT ETHERNET CONTROLLER

A TT Ethernet node consist of a TT Ethernet controller and a host computer. The data exchange interface between the host computer and the TT Ethernet controller is denoted as *communication network interface* (CNI). The CNI is implemented as dual ported memory, which serves as temporal firewall between the host and the communication controller [12].

The design consists of: the *transmitter blocks*, the *receiver blocks*, the *timer module*, the *configuration modules* and the *control module*.

The structure of the design of a TT Ethernet controller is depicted in Figure 1. The Communication Network Interface (CNI) and the host interface are not covered by the design of the TT Ethernet controller because they depend on the hardware used for the implementation of the TT Ethernet controller.

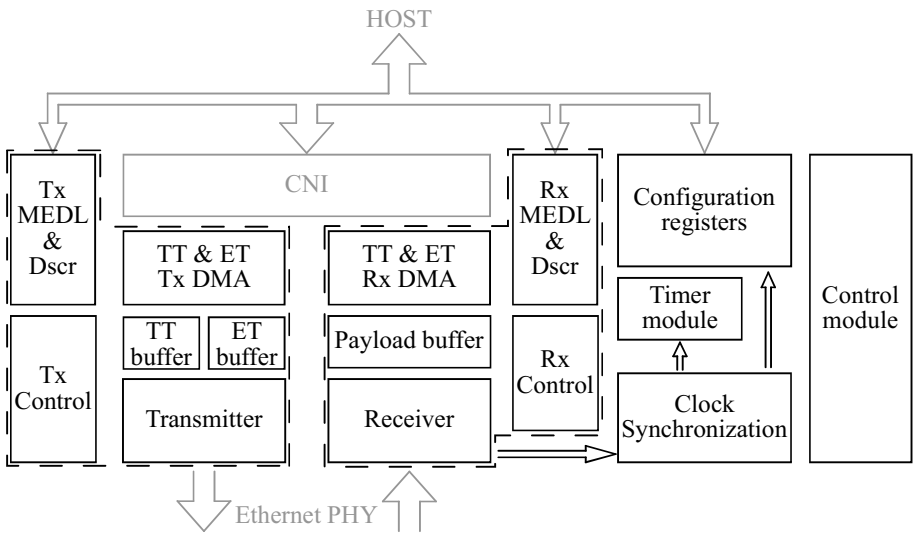


Figure 1. Structure of the TT Ethernet Controller Design

The Control Module

The *control module* coordinates the functions of the different components for TT message handling. ET message handling can be enabled by the host at any time, independent of the state of the control module. The controllers internal state equals to one of the protocol states specified at the TT Ethernet protocol specification [13].

There are two possible startup scenarios for a controller:

1 In case it is configured to be a *clock rate-master node*, the controller waits a certain period of time for an incoming synchronization message. If a synchronization message is received, another controller acts as primary rate master, and this controller has to behave as backup rate master, synchronizing its clock to the primary rate master until it fails. If no synchronization message can be received at startup, this controller becomes the primary rate-master node. Then the controller begins to transmit TT messages as configured, starting with a synchronization message.

2 If the controller is configured as *timekeeping node*, it adjust its local time to the clock of the rate master. At startup, the controller waits until it receives enough synchronization messages so that the local time can be synchronized. If a certain precision has been reached, the controller starts to transmit TT messages as configured.

Configuration Modules

The *configuration modules* consists of two components: the configuration registers and the MEDL memory.

The *configuration registers* are static and are set by the host before enabling the message transfer of the controller. During runtime, the host can access several read-only status registers to be informed about the current controller operational status.

The schedule information of each packet is stored in the *Message Descriptor List (MEDL) memory*. There is one MEDL memory for receiving and one for transmitting messages. The MEDL is organized in 16 linked lists (TT Ethernet supports periodic transmission of messages with 16 different periods). Each MEDL record contains a *MEDLPTR* entry, which is used to link to the next MEDL record. In each list the entries are sorted with rising transmission offset. If a new entry is inserted into the linked list the host has to make sure that it is inserted at the right position within the list, the actual memory location of the entry is not of importance. An incorrectly sorted list would result in severe schedule errors. A DMA mechanism autonomously stores status information about the actual transmission and fetches the next entry of each list after the actual message transfer has finished. To gain concurrent access to the next messages of each linked list, up to 16 active MEDL entries are duplicated inside of the controller.

The host stores the MEDL data in a dual ported RAM. Only memory with two independent ports allows the simultaneous access for the host and the controller

to the MEDL data. There is only one exception; it has to be ensured that a single memory location is only written by one party at a time. If the message transmission is enabled, the controller has a fixed access schema for the MEDL so the host cannot change a MEDL entry at any time. The host has to follow a schedule when it is allowed to modify a certain MEDL entry. Reading the MEDL entries is allowed at any time, however, the read operation of the host is not atomic and may produce inconsistent data. This problem is solved by deploying the non-blocking-write protocol [14].

TT data transmission

After the controller is configured, the TT message data transmission is done autonomously. If the transmission time has arrived, the controller starts to transmit the message header. At the same time one of the controller's DMA engines transfers the packet's payload data from the CNI memory to an internal buffer memory. After the header transmission has finished, the controller starts to read data from the internal buffer and transmits it after the header, until the final byte is transmitted out of the buffer. If more then 59 bytes are sent, the minimum message length as defined by the IEEE 802.3 standard is met so the packets checksum is transmitted and the operation has finished. If less than 60 bytes are sent the controller adds padding bytes until the required 60 bytes are transmitted before adding the checksum.

The checksum is generated on-the-fly while the data is sent. So it is impossible for the host to void a pre-calculated checksum by changing data while the controllers DMA is active.

TT Ethernet distinguishes between different kinds of TT packets: TTsync, TTdiag and TT data messages. The difference between TTsync and TTdiag on the one hand and TT data messages on the other hand, is that only TT data messages have payload data which is stored in the CNI memory. Therefore it is possible for the sender module to generate TTsync and TTdiag messages on-the-fly without the use of a DMA mechanism. This decreases the processing time of these messages and enables the generation of packets with the shortest possible inter-packet gap.

The schedule information of each packet is stored in the MEDL memory. The schedule information is the basis for the trigger generator, which signals when each message transmission time has arrived. With this signal, the TX DMA is started (if a TT data message is scheduled) and the sender starts to transmit the packet. If the whole packet is sent, the outcome of the message transmission is updated in the status fields of the MEDL entry and the next MEDL entry of this message period is loaded.

TT data reception

The data reception is similar to the data transmission. The receiver transfers the data into an internal buffer while checking the validity. As the validity of the data can only be determined after the whole packet is examined and the frame checksum is compared, the DMA which copies the data into the CNI memory is started after the whole packet is received. If the packet was invalid, all the data is cleared from the buffer, and only the status information is updated in the CNI memory.

Synchronization messages get a special treatment by the receiver. Additional to the internal buffer, some content of the payload-data and the reception time-stamp are also written into a FIFO buffer which is connected to the clock synchronization interface.

Other than the transmission, the reception needs a validity window with configurable size, because all nodes in a cluster cannot be synchronized perfectly, and this is compensated by this reception window. If a message is received within the bounds of this window, it is treated as valid in terms of reception time. If no message is received within the reception window, the message associated to this window is treated as lost and its status is changed accordingly. The size of this reception window is configurable and chosen according to the quality of the clock synchronization within the cluster. The better the clocks are synchronized among each other, the smaller the size of the reception window can be configured.

The message transmission delay between two TT Ethernet controllers caused by the network is constant. This delay depends on the number of switches and the length of the cables in the network.

In order to simplify the clock synchronization mechanism, the TT Ethernet controller adjusts each time-stamp taken for any incoming message by the message transmission delay. These values have to be configured by the host before the controller is enabled.

ET message handling

If the TT part of the controller is enabled, ET data transmission is done in between two TT messages, when the network is idle. If TT message handling is disabled by the host, ET messages can be transmitted at any time. If the host configures an ET message for transmission, the controller waits for an inter-packet-gap between two scheduled TT messages which is big enough for the ET message to be sent. The controller will not change the order of ET packets if there are several messages to be sent. So it is guaranteed that the receivers

don't have to reorder the packets. This is necessary as especially embedded nodes don't have enough reception buffer space to hold several packets. They depend on the right reception order or they drop all packets which are out of order and request re-transmissions. This would waste network bandwidth, and so it is avoided in this design.

The reception of ET messages follow the same procedure as TT message reception except that ET messages do not have to be received within a receive-time window.

Timer Module

The timer module provides several counters. One of them is used to implement a configurable frequency divider. It is used to apply rate correction in order to adjust the local clocks frequency to the frequency of the global time of the cluster. This counter is decremented with each Microtick. The Microtick frequency is defined by the used hardware oscillator. Each time this counter reaches zero, a pulse (the Macrotick) is generated and it is reloaded with a value computed by the clock synchronization subsystem. The Macrotick pulse is used to increment the local time counter. By changing the length of a Macrotick period, the rate of the local clock can be adjusted.

The timer module provides an interface for the clock synchronization subsystem. The actual computation of the synchronization algorithm is done using a CPU. This has the advantage, that the algorithm can be implemented in software. So it is easy to apply changes to the clock synchronization algorithm.

In case of an incoming TTsync message, the receiver stores the reception time stamp and the timing data included in the message into a buffer. The content of this buffer can be read by the CPU subsystem which uses this information to compute the difference between the local time and the time to synchronize to. It performs the clock state correction such that the actual time difference is compensated and calculates and applies a new Microtick/Macrotick rate value, such that this time difference is reduced until the next TTsynch message will be received. In the case that the time difference is larger than 1 Macrotick, the clock synchronization module will restart and set the local time to the time obtained from the clock synchronization message.

When, for a certain amount of time, the time difference is smaller than a Macrotick length, the rate and time value of the local clock is adjusted to the reference clock and it signaled that the clocks are synchronized and the message transmission can be enabled.

4. TTE CONTROLLER PROTOTYPE

An overview of our hardware designed for a TTE controller implementation is shown in Figure 2.

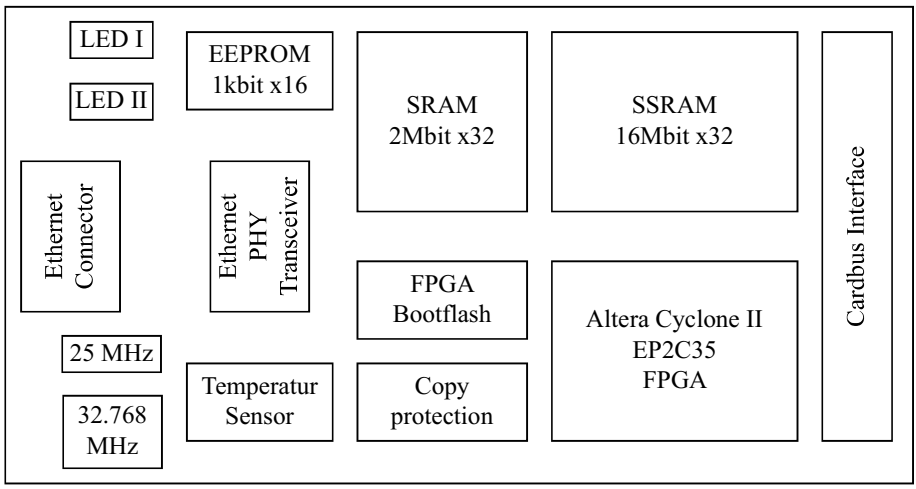


Figure 2. TTE Controller Card Overview

The TTE Controller Card interfaces to the host via a Cardbus interface. The configuration and data area of the card are mapped into the host computers memory space.

All the functionality of the card is implemented in the FPGA, an EP2c35 Altera Cyclone II device [15]. The Ethernet PHY transceiver is used to convert the IO standard of the FPGA to the Ethernet physical level standard. The 2 MB sized SSRAM memory holds the CNI memory for real-time messages as well as buffer space for standard Ethernet messages. The non-volatile memories are holding the configuration data of the FPGA (which bases on volatile SRAM cells and needs therefore to be configured when powered up) and some configuration data, which is unique to each card and needed to identify the device on the network and by the host. A sensor which can be used to monitor the temperature of the card and shutdown the system in case of overheating is also implemented as well as a copy protection circuit to prevent theft of the used IPs. The 25MHz Oscillator is needed by the Ethernet PHY; the FPGA uses the 32.768MHz oscillator. Status information of the controller is shown by the two dual-color LEDs.

The TT Ethernet controller was included in a SOPC system applied to the FPGA. Table 1 shows the compilation results of the TT Ethernet controller module excluding the clock synchronization CPU and the host interface.

Table 1. Compilation results of the TT Ethernet controller module

Size	18598 Logic Cells
Memory usage	144896 bits
Maximum Microtick frequency	70.63 MHz

5. EXPERIMENTAL EVALUATION

The hardware setup consisted of a TT Ethernet switch as described in [16] which was connected to three nodes equipped with a TT Ethernet controller as described in Section 4. Node 0 was configured to be the rate master node, Node1 and 2 were timekeeping nodes. A fourth node equipped with a COTS Ethernet controller and running Windows XP was connected to generate ET packets.

Minimum Inter-Packet-Gap

This experiment has been performed to show that the system can handle packet bursts with a minimum inter-packet-gap of 3 Macroticks. Node 0 was configured to transmit packets according to the schedule listed in Table 2 and Node 1 received all of the messages. Node 2 and the ET packet generator node were disconnected during the experiment.

Table 2. TT message schedule used at this experiment

type	*length*	*period*	*offset*	*duration*
TTsync	64 Bytes	7,8ms	0,00μs	5,80μs
TTpM	666 Bytes	7,8ms	7,60μs	54,00μs
TTpM	122 Bytes	7,8ms	62,94μs	10,48μs

The schedule listed in Table 2 creates message bursts with an inter-packet-gap between the two data messages of 1.34μs which is equal to 2.75 Macroticks. The host CPU on Node 1 was configured to monitor the packet reception. After 10000 transmission periods, the data recorded by Node 1 showed that each packet was received without errors and in time.

Clock Synchronization

This experiment deals with the synchronization of the local clocks within the nodes to the global time of the cluster. For these experiment a simple *master-slave synchronization algorithm* was used. The cluster consisted of three TT Ethernet nodes whereas one node is configured as rate-master node, sending TTsync messages with a period of 7.8ms. The other two nodes were timekeeping nodes, which synchronized their local clocks to the clock of the rate-master. Each of the three nodes was configured to transmit and receive at least one packet per period. The transmission offset of the data messages was constant; the period was changed in the course of the experiment (see Table 3).

Table 3. Used message scheduling of the experiments

Period	*Offset*	*Sender*	*Receiver*
for all experimental runs :			
7.8ms	0	Node 0	Node 1, Node 2
for experimental run (x) $0-3$ **:**			
61μs - 488μs	2^{-16+x}	Node 0	Node 1, Node 2
61μs - 488μs	2^{-15+x}	Node 2	Node 0, Node 1
61μs - 488μs	$2^{-15+x} + 2^{-16+x}$	Node 1	Node 2
for experimental run (x) $4-15$ **:**			
976μs - 2s	$2^{-16+x} + 2^{-23+x}$	Node 0	Node 1, Node 2
976μs - 2s	$2^{-15+x} + 2^{-23+x}$	Node 2	Node 0, Node 1
976μs - 2s	$2^{-15+x} + 2^{-16+x} + 2^{-23+x}$	Node 1	Node 2

The experiment was repeated 32 times. At the first run, the data messages were transmitted using a period of 61μs. Then the period was doubled at each subsequent run. After 16 runs Node 4, a standard PC computer with a COTS Ethernet controller running Windows XP, was enabled to send one ET message with a length of 1500 bytes every 250μs and the period was reset to 61μs to repeat the first 16 runs with enabled ET traffic.

For each experimental run, all nodes measure the receive time deviation of each message from the scheduled arrival time of the message. As each message is only accepted if it is received within a predefined reception window, the reception time deviation of a message is of particular importance. The maximum measured deviation of all experimental runs are summarized at Table 4.

As the measurement results listed in Table 4 are showing, the receive time deviation with ET traffic enabled is the same as without ET traffic, the maxi-

Table 4. Measured time deviations of arriving data messages

	Node 0 earliest	*Node 0 latest*	*Node 1 earliest*	*Node 1 latest*	*Node 2 earliest*	*Node 2 latest*
max. deviation without ET	-9μT	9μT	-9μT	8μT	-7μT	7μT
max. deviation with ET	-6μT	11μT	-5μT	8μT	-5μT	8μT

mum measured receive time deviations of the messages are still below half a Macrotick (25μT), so in both experiments there were no lost messages.

6. CONCLUSION

A node within a TT Ethernet system consists of a host computer and a communication controller. The host computer executes one or more application tasks, while the communication controller is in charge of executing the TT Ethernet communication protocol. The host and the communication controller are connected via the Communication Network Interface (CNI). The TT Ethernet communication controller can handle both, ET and TT messages.

There are several ways to implement a communication controller for TT Ethernet:

- In software as additional task running on the host computer
- As software running on a dedicated CPU
- As a dedicated hardware device

In this paper a design of a TT Ethernet controller is introduced, which is optimized for implementation in hardware. For the evaluation of the design of the TT Ethernet controller, a prototype implementation was developed.

A TT Ethernet system consisting of a TT Ethernet switch and several TT Ethernet nodes was evaluated in the course of this work. The achievable precision of the system was measured first. One experimental run for each possible message transmission period defined by the TT Ethernet specification was made. The measurements showed a worst case precision of 18μT or 342 ns. Then the experiments where repeated applying ET traffic to the network. The measurements of these experiments show the same result.

The last experiment was executed to show that the TT Ethernet controller is able to handle TT packet bursts with a minimum inter-packet-gap of 3 Macroticks.

By analyzing the evaluation experiments it can be concluded that a design which applies the mechanisms of TT Ethernet and which is carefully implemented is able to provide a message transmission environment which exhausts the limits of the Ethernet technology while still being able to support standard Ethernet devices without changes at any device within the cluster.

REFERENCES

[1] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9894-7.

[2] IEEE Standard 802.3, 2000 Edition. Carrier Sense Multiple Access with Collision Detect on (CSMA/CD) Access Method and Physical Layer Specifications., 2000.

[3] Bernard Jouga. How Ethernet becomes Industrial. *Workshop: The use of Ethernet as a fieldbus*, September 2001.

[4] Chitra Venkatramani and Tzi-cker Chiueh. Supporting Real-Time Traffic on Ethernet. In *Real-Time Systems Symposium (RTSS)*, pages 282–286, December 1994.

[5] Seok-Kyu Kweon and Kang G. Shin. Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing. In *Sixth IEEE Real-Time Technology and Applications Symposium*, pages 90–100, May 2000.

[6] Jean-Dominique Decotignie. Ethernet-Based Real-Time and Industrial Communications. *Proceedings of the IEEE, Volume: 93, Issue: 6*, 2005.

[7] Max Felser. Real-Time Ethernet-Industry Prospective. *Proceedings of the IEEE, Volume: 93, Issue: 6*, pages 1118–1129, June 2005.

[8] Paulo Pedreiras, Luis Almeida, and Paolo Gai. The FTT-Ethernet Protocol: Merging Flexibility, Timeliness and Efficiency. In *14th Euromicro Conference on Real-Time Systems*, June 2002.

[9] Website: Information about Real-Time Ethernet in Industry Automation. $http://www.Real-Time-Ethernet.de$, 2004. Online; Accessed 19-October-2006.

[10] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. The Time-Triggered Ethernet (TTE) Design. *8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), Seattle, Washington*, May 2005.

[11] Astrit Ademaj, Hermann Kopetz, Petr Grillinger, Klaus Steinhammer, and Alexander Hanzlik. Fault-tolerant time-triggered ethernet configuration with star topology. *Dependability and Fault Tolerance Workshop*, Mar. 2006.

[12] Hermann Kopetz. The CNI: A Generic Interface for Distributed Real-Time Systems. Confidential draft, Technische Universitat Wien, Institut fur Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1998.

[13] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. Time-Triggered Ethernet Protocol Specifcation. currently unpublished, Technische Universitat Wien, Institut fur Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2006.

[14] Hermann Kopetz and Johannes Reisinger. NBW: A Non-Blocking Write Protocol for Task Communication in Real-Time Systems. In *Proceedings of the 14th Real-Time Systems Symposium*, pages 290–299, Raleigh-Durham, North Carolina, USA, December 1993.

[15] Altera Corp. *Cyclone Device Handbook*. $http://www.altera.com/literature/hb/cyc/cyc_c5v1.pdf$, August 2005. Online; Accessed 19-October-2006.

[16] Klaus Steinhammer, Petr Grillinger, Astrit Ademaj, and Hermann Kopetz. A Time-Triggered Ethernet (TTE) Switch. *Design, Automation and Test in Europe, Munich, Germany*, March 2006.

ERROR CONTAINMENT IN THE TIME-TRIGGERED SYSTEM-ON-A-CHIP ARCHITECTURE

R. Obermaisser, H. Kopetz, C. El Salloum, B. Huber
Vienna University of Technology, Austria

Abstract: The time-triggered System-on-a-Chip (SoC) architecture provides a generic multi-core system platform for a family of composable and dependable giga-scale SoCs. It supports the integration of multiple application subsystems of different criticality levels within a single hardware platform. A pivotal property of the architecture is the integrated error containment, which facilitates modular certification, robustness, and composability. By dividing the complete SoC into physically separated components that interact exclusively by the timely exchange of messages on a time-triggered Network-on-a-Chip (NoC), we achieve error containment for both computational and communication resources. The time-triggered design allows protecting the access to the NoC with guardians that are associated with each component. Based on the protection of the time-triggered NoC with inherent predictability and determinism, the architecture also enables error containment for faulty computational results. These value message failures can be masked using active redundancy (e.g., off-chip and on-chip Triple Modular Redundancy (TMR)) or detected using diagnostic assertions on messages. The design of the error containment mechanisms systematically follows a categorization of significant fault classes that an SoC is subject to (e.g., physical/design, transient/permanent). Evidence for the effectiveness of the error containment mechanisms is available through experimental data from a prototype implementation.

1. INTRODUCTION

Many large embedded control systems can be decomposed into a number of nearly independent *Distributed Application Subsystems (DASes)*. In the automotive domain, the *power train control system*, the *airbag control system*, the *comfort electronics control system* and the *multimedia system* are

Obermaisser, R., Kopetz, H., El Salloum, C., Huber, B, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 339–352.

examples for DASes. A similar decomposition is performed in control system design aboard an airplane. Different DASes can be of *differing criticality level* and are often developed by *different organizations*. At a high level of abstraction—at the *Platform Independent Model (PIM)* level—a DAS can be described by a set of processing *Jobs* that exchange *messages* in order to achieve its stated objective. In order to eliminate any *error propagation path* from one DAS to another DAS and to reduce the overall system complexity, each DAS is often implemented on its own dedicated hardware base, i.e., a computer is assigned to each job of a DAS and a shared physical communication channel (in the automotive domain a *CAN* network (Bosch 1991)) is provided for the exchange of the messages within a DAS. In case of a failure of a DAS function it is then straightforward to identify the organization responsible for the malfunction, even if it is not clear whether the failure is caused by a transient hardware fault or a software error. We call such an architecture, where each DAS has its own dedicated hardware base, a *federated architecture.* In the automotive domain the massive deployment of federated architectures has led to a large number of Electronic Control Units (ECUs, i.e., nodes) and networks aboard a car. In a typical premium car more than fifty ECUs and five different networks can be found (Leohold 2005). This large number of ECUs and networks has some negative consequences: the high number of cabling contact points (which are a significant cause of failures) and the high costs. These negative consequences could be eliminated if one ECU could host more than one job of a DAS and thus the number of ECUs, networks and cables is significantly reduced. We call such an architecture, where a single integrated hardware base for the execution of different DASes is provided, an *integrated architecture*. Hammett R. describes aptly the technical challenge in the design of an integrated architecture: *The ideal future avionics systems would combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware efficiency benefits of an integrated system (Hammett 2003).*

In the recent past, a number of efforts have been made to develop an integrated architecture e.g., *Integrated Modular Avionics (IMA) (Wilkinson 2005)* in the aerospace domain, AUTOSAR (Heinecke et al. 2004) in the automotive domain, and DECOS (Obermaisser et al. 2006) as a cross-domain architecture. The key idea in these approaches is the provision of a partitioned operating system for a computer with a single powerful CPU. This operating system is intended to provide in each partition an encapsulated execution environment for a single job and eliminate any error propagation path from one job to another job. However, the required encapsulation, particularly w.r.t. to temporal properties and transient failures is difficult to achieve in such an architecture.

This paper follows another route. The recent appearance *of multi-core Systems-on-a-Chip (SoCs)* (e.g., the *Cell multiprocessor (Kahle, Day et al. 2005)),* makes it possible to achieve the desired integration by assigning each job to a core of an SoC and by providing a time-triggered on-chip interconnect that supports composability and error containment between DASes. This paper focuses on a key property of this architecture, the *error containment between DASes*. The paper is structured as follows: In Section two we present an overview of the time-triggered SoC architecture. Section three is devoted to the issues of error containment with respect to design faults. Section four deals with error containment with respect to physical faults. Section five discusses implementation aspects and an experimental evaluation of the architecture. The paper terminates with a conclusion in Section six.

2. TIME-TRIGGERED SOC ARCHITECTURE

The central element of the presented SoC architecture is a *time-triggered NoC* that interconnects multiple, possibly heterogeneous IP blocks called *micro components* (see Figure 1), each one hosting a job of a DAS. The SoC introduces a *trusted subsystem*, which ensures that a fault (e.g., a software fault) within the host of a micro component cannot lead to a violation of the micro component's temporal interface specification in a way that the communication between other micro components would be disrupted. Therefore, the trusted subsystem prevents a faulty micro component from sending messages during the sending slots of any other micro component.

Another focus of the SoC architecture is integrated support for maintenance. The *diagnostic unit* is an architectural element that executes assertions on the messages sent by the micro components and stores diagnostic information in persistent storage for a later analysis.

Furthermore, the time-triggered SoC architecture supports dynamic integrated resource management. For this purpose, a dedicated architectural element called the *Trusted Network Authority (TNA)* accepts run-time resource allocation requests from the micro components and reconfigures the SoC, e.g., by dynamically updating the time-triggered communication schedule of the NoC and switching between power modes of micro components.

2.1 Micro Component

The introduced SoC can host jobs of multiple DASes (possibly of different criticality levels), each of which provides a part of the service of

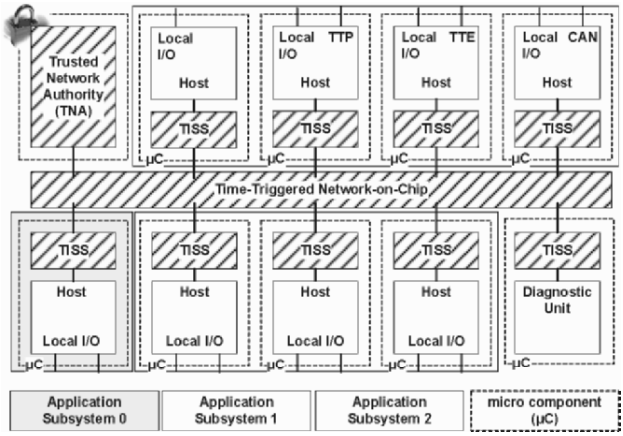


Figure 1: Structure of Time-Triggered SoC Architecture: trusted subsystem (shaded) and non trusted subsystem (hosts of micro components)

the overall system. A nearly autonomous IP-block, which is used by a particular DAS is denoted as a micro component. A micro component is a self-contained computing element, e.g., it can be implemented as a general purpose processor with software, and FPGA or as special purpose hardware. A DAS can be realized on a single micro component or by using a group of possibly heterogeneous micro components (either located on one or on multiple interconnected SoCs).

The interaction between the micro components of an application subsystem occurs solely through the exchange of messages on the time-triggered NoC. Each micro component is encapsulated, i.e., the behavior of a micro component can neither disrupt the computations nor the communication performed by other micro components. For this purpose, each micro component contains a so-called Trusted Interface Subsystem (TISS), which guards the access of the micro component to the time-triggered NoC (see also Section 3).

Encapsulation prevents *by design* temporal interference (e.g., delaying messages or computations in another micro component) and spatial interference (e.g., overwriting a message produced by another micro component). The only manner, in which a faulty micro component can affect other micro components, is by providing faulty input value to other micro components of the application subsystem via the sent messages.

Due to the provided encapsulation, the SoC architecture supports the detection and masking of such a value failure of a micro component using Triple Modular Redundancy (TMR). Encapsulation is necessary for ensuring the independence of the replicas. Otherwise, a faulty micro component could disrupt the communication of the replicas, thus causing common mode failures.

Encapsulation is also a key mechanism for the *correctness-by-construction* of application subsystems on an SoC. The SoC architecture ensures that upon the incremental integration of micro components, the prior services of the already existing micro components are not invalidated by the new micro components. This property, which is denoted as *composability* (Kopetz and Obermaisser 2002), is required for the seamless integration of independently developed DASes and micro components.

Also, encapsulation is of particular importance for the implementation of SoCs encompassing DASes of different criticality levels. Consider for example a future automotive system, which will incorporate DASes ranging from a safety-critical drive-by-wire DAS to a non safety-critical comfort DAS. In such a mixed criticality system, a failure of micro components of a non safety-critical application subsystem must not cause the failure of application subsystems of higher criticality.

2.2 Time-Triggered Network-on-a-Chip

The time-triggered NoC interconnects the micro components of an SoC. The purposes of the time-triggered NoC encompass clock synchronization for the establishment of a global time base, as well as the predictable transport of periodic and sporadic messages.

Clock Synchronization: The time-triggered NoC performs clock synchronization in order to provide a global time base for all micro components despite the existence of multiple clock domains. The time-triggered NoC is based on a uniform time format for all configurations, which has been standardized by the OMG in the smart transducer interface standard (OMG 2002).

Predictable Transport of Messages: Using Time-Division Multiple Access (TDMA), the available bandwidth of the NoC is divided into periodic conflict-free sending slots. We distinguish between two utilizations of a periodic time-triggered sending slot by a micro component. A sending slot can be used either for the periodic or the sporadic transmission of messages. In the latter case, a message is only sent if the sender must transmit a new event to the receiver. If no event occurs at the sender, no message is sent and thus no energy is consumed.

2.3 Diagnostic Unit

The diagnostic unit helps maintenance engineers in choosing appropriate maintenance actions. A maintenance action is either an update of the software in the host of a micro component to eliminate a design fault, or the replacement of a SoC component that is subject permanent physical faults.

For this purpose, the diagnostic unit collects error indications from micro components and also performs error detection itself. For each detected error, an entry is inserted into an error log for a later analysis.

Table 1: DSoS Message Classification

Attribute	***Interface Feature Check***
Valid	A message is valid if it meets CRC, range and logical checks
Checked	A message is checked if it passes the output assertion
Permitted	A message is permitted w.r.t. a receiver, if it passes the input assertion of that receiver
Timely	A message is timely if it is in agreement with its temporal specifications
Correct	A message is correct if its value and temporal specifications are met
Insidious	A message is insidious if it is permitted but incorrect (requires a global judgment)

The error detection is based on the message classification defined in the Dependable Systems-of-Systems (DSoS) conceptual model (Jones, Kopetz et al. 2001). In the DSoS model (see *Table 1*), a message is classified as *checked,* if it passes the *output assertion*. In addition, each message has to pass an *input assertion* in order to be processed by the receiving micro component. Once a message passes the input check it is called *permitted*. The input and output assertions consist of syntactic, temporal, and semantic checks. A message is called *valid,* if it passes the syntactic check. The CRC ensures that the content is in agreement with the checksum. We term a message *timely* if it is in agreement with the temporal specification. The *value-correctness* of a message can only be fully judged by an omniscient observer. However, application-specific plausibility checks can be applied. Note, that this implies the possibility that a message is judged as being *permitted* and therefore passing the input assertion but classified as incorrect by the omniscient observer. Such a message is called *insidious*.

In order to perform this message classification, the following three types of checks are executed at the SoC:

1. ***Output assertions***: The output assertions are computed at the diagnostic unit. The diagnostic unit observes all messages that are exchanged on the time-triggered NoC and executes predicates encoding a priori knowledge w.r.t. to the message syntax and semantics.
2. ***Temporal checks***: For sporadic communication, the TISS detects overflows of message queues. These checks apply both to queues with received messages (i.e., input ports), as well as to queues with messages that shall be sent (i.e., output ports).

3. ***Input assertions***: Input assertions are computed by micro components that receive messages from other micro components. Input assertions test the message syntax and the message semantics.

Errors detected via these three mechanisms are stored in the error log. While the results of (1) are already available at the diagnostic unit, the checks in (2) and (3) employ error indication messages on a diagnostic communication channel. The diagnostic communication channel of a micro component is realized as a statically reserved slot of the time-triggered NoC.

3. ERROR CONTAINMENT FOR DESIGN FAULTS

In his classical paper (Gray 1986), Jim Gray proposed to distinguish between two types of software design errors, Bohrbugs and Heisenbugs. Bohrbugs are design errors in the software that cause reproducible failures. Heisenbugs are design errors in the software that seem to generate quasi-random failures. From a phenomenological point of view, a transient failure that is caused by a Heisenbug cannot be distinguished from a failure caused by transient hardware malfunction.

In the SoC architecture, an error that is caused by a design fault in the sending micro component could only propagate to another micro component via a message failure, i.e., a sent message that deviates from the specification. In general, a message failure can be a *message value failure* or a *message timing failure (Cristian and Aghili 1985)*. A message value failure implies that a message is either invalid or that the data structure contained in a valid message is incorrect. A *message timing failure* implies that the *message send instant* or the *message receive instant* are not in agreement with the specification.

The propagation of an error due to a message timing failure is prevented by design in the SoC architecture. The TISS acts as a guardian for a micro component and prevents that a micro component can violate its temporal specification. In contrast, the value failure detection is in the responsibility of the receiving micro components and the diagnostic unit of the SoC architecture.

In order to avoid the propagation of an error caused by a message failure we need error detection mechanisms that are in different Fault Containment Regions (FCRs) than the message sender. Otherwise, the error detection mechanism may be impacted by the same fault that caused the message failure.

3.1 Fault Containment Regions for Design Faults

The FCRs of the SoC architecture with respect to design faults are as follows:

Trusted Subsystem: We assume that the trusted subsystem is free of design faults. In order to justify this strong assumption, the design of the trusted subsystem must be made as small and simple as possible in order to facilitate formal analysis. The time-triggered design of the trusted subsystem reduces the probability of Heisenbugs (e.g., no race conditions, clear separation of logical and temporal control). The control signals are derived from the progression of the sparse global time base (Kopetz 1992), which guarantees that all TISSes will visit the same state within a silence interval of the sparse time base.

Micro Components: The non safety-critical software of a micro component is encapsulated by the trusted subsystem such that even a malicious fault in the non safety-critical software of the host will not affect the correct function of the safety-critical software in the hosts of other micro components.

3.2 Timing Failure Containment

For the purpose of error containment w.r.t. message timing failures, each micro component comprises two parts: a host and a Trusted Interface Subsystem (TISS). The host implements the application services. Using the TISS, the time-triggered SoC architecture provides a dedicated architectural element that protects the access to the time-triggered NoC. Each TISS contains a table which stores a priori knowledge concerning the global points in time of all message receptions and transmissions of the respective micro component. Since the table cannot be modified by the host, a design fault restricted to the host of a micro component cannot affect the exchange of messages by other micro components.

3.3 Value Failure Containment

Since value failures are highly application-specific, their detection in the SoC architecture relies on the host. An important baseline for these detection mechanisms is the timing failure containment of the time-triggered NoC. The SoC architecture ensures that a received message truly stems from the associated sender (i.e. masquerading protection) and that the message is temporally correct.

It is the responsibility of each micro component, to decide whether the actual message should be accepted or not. Therefore, it can run tests

exploiting application-specific knowledge concerning the values of the signals (variables) that are transported within the message. Such tests are usually called executable assertions (Hecht 1976; Saib 1987). Executable assertions are statements, which can be made about the signals within a message. They are executed whenever a message is received in order to see if they hold. If not, the message is regarded as incorrect and will be discarded.

In the SoC architecture we support various types of assertions. For example executable assertions with signal range checks and slew rate checks can be applied to check whether a value is within its physical limits (e.g., the speed of a car cannot change within one second from zero to 200 kilometers per hour). The combination of signal range checks and slew rate checks into a single assertion offers the possibility to define bounds for extreme values in dependency on the actual rate of change.

A more elaborated technique is "model-based fault detection" (Isermann, Schwarz et al. 2002), where a process model of the controlled system is used to detect errors by relating two or more system variables to each other.

4. ERROR CONTAINMENT FOR PHYSICAL FAULTS

Depending on the persistence of a fault, one can classify physical faults into permanent faults and transient faults (Avizienis, Laprie et al. 2001). A permanent fault occurs, if the hardware of an FCR brakes down permanently. An example for a permanent fault is a stuck-at-zero fault of a memory cell. A transient fault is caused by some random event. An example for a transient fault is a Single Event Upset (SEU) caused by a radioactive particle (Normand 1996).

4.1 Fault-Containment Region for Physical Faults

In non safety-critical systems, we distinguish the same types of FCRs on the SoC architecture for physical faults as for design faults, namely the trusted subsystem and individual micro components. For a physical fault that affects a micro component, the SoC is capable of recovering from its induced errors. The performed recovery strategy depends on the persistence of the fault.

A permanent fault affecting a micro component requires reallocating the functionality of the affected micro component to a correct one. If the SoC contains multiple micro components of identical hardware, where the functional differentiation is provided by application software, the allocation of application subsystems to micro components can be dynamically

reconfigured by the TNA. Relaxing the requirement for 100% correctness for devices and interconnections may dramatically reduce the cost of manufacturing, verification and test of SoCs. However, a permanent fault affecting the trusted subsystem is not tolerated by the SoC architecture. For transient faults, on the other hand, a restart of the micro component can be triggered.

Contrary to non safety-critical systems, for safety-critical systems we have to consider the entire SoC as a single FCR. Due to the shared power supply, physical components (e.g. housing), and the proximity we cannot assume with a probability demanded for, e.g., ultra dependable systems that a physical fault like a SEU affects only a single micro component while sparing the trusted subsystem of the SoC. Therefore, masking of physical faults for safety-critical systems has to be performed at cluster level, where multiple SoC are interconnected to a distributed system.

4.2 Non Safety-Critical Systems

The masking of timing failures occurs in the same manner as for design faults. In case the host of a micro component is affected by a physical fault, the trusted subsystem ensures that there is no interference with the timely exchange of messages by other micro components.

Value failure detection is not in the responsibility of the SoC architecture, but in the responsibility of the micro components. For example, in non safety-critical systems value failure detection and correction can be performed in a single step by on-chip triple modular redundancy (TMR). In contrast to design faults, the masking of value failures resulting from physical faults can be performed in a systematic application-independent manner through active redundancy such as TMR. In this case, three replicated deterministic micro components perform the same operations in their host. They produce – in the fault-free case – correct messages with the same content that are sent to three replicated receivers that perform a majority vote on these three messages.

Value failure detection and timing failure detection are not independent. In order to implement a TMR structure for value failure masking at the application level in the micro components, the integrity of the timing of the architecture must be assumed. An intact sparse global time-base is a prerequisite for the system-wide definition of the distributed state, which again is a prerequisite for masking value failures by voting.

4.3 Safety-Critical Systems

In ultra-dependable systems, a maximum failure rate of 10^{-9} critical failures per hour is demanded (Suri, Walter et al. 1995). Today's technology does not support the manufacturing of chips with failure rates low enough to meet these reliability requirements. Since component failure rates are usually in the order of 10^{-5} to 10^{-6} (e.g., (Pauli et al. 1998) uses a large statistical basis and reports 100 to 500 failures out of 1 Million ECUs in 10 years), ultra-dependable applications require the system as a whole to be more reliable than any one of its components. This can only be achieved by utilizing fault-tolerant strategies that enable the continued operation of the system in the presence of component failures. Therefore, it is demanded to realize a distributed system based on the SoC architecture. For this purpose, the SoC architecture supports gateways for accessing chip-external networks (e.g., TTP (Kopetz and Grünsteidl 1994), TTE (Kopetz, Ademaj et al. 2005)).

By the use of a time-triggered protocol (e.g. TTP or TTE) for the chip external network, timing failure containment can be realized by a guardian, which judges a message as untimely based on a priori knowledge of the send and receive instants of all messages of a single SoC.

As for non-safety-critical systems, value failure containment can be implemented by the use of TMR. However, for safety-critical systems the three replicated micro components have to reside on different micro components.

5. IMPLEMENTATION

Using a distributed system, the prototype implementation emulates an SoC with four micro components, as well as the TNA. A Time-Triggered Ethernet (TTE) network emulates the NoC and supports the time-triggered exchange of periodic and sporadic messages.

5.1 Implementation of Micro Components

Each micro component consists of two single board computers, namely one implementing the host and the second one implementing the TISS. The single board computer of the host is a Soekris Engineering net4521, which incorporates a 133Mhz 468 class ElanSC520 processor from AMD, 64 MBytes of RAM, and two 100 Mbit Ethernet ports. The implementation of the TISS is based on a single board compact computer of type Soekris Engineering net4801, which incorporates a 266 MHz 586 class NSC SC1100

processor. As an operating system, both single board computers use the realtime Linux variant Real-Time Application Interface (RTAI) (Beal, E. Bianchi et al. 2000).

In our prototype implementation we emulate the time-triggered NoC with TTE (Kopetz, Ademaj et al. 2005). For this purpose, a hardware implementation of the TTE controller is employed which is based on a PCMCIA FPGA card.

Dynamic reconfiguration of the NoC occurs via configuration messages sent by the TNA to the TISSes. The reconfiguration is triggered by the TTE controller via a dedicated reconfiguration interrupt at a predefined periodic instant with reference to the global time. Thus the reconfiguration is consistently performed at all TISSs at the same tick of the global time. At the reconfiguration instant, the TTE controller is switched to inactive and the MEDL is updated. The controller is reactivated after the update has finished.

The prototype implementation implements the memory interface between host and TISS via a standard Ethernet connection. On top of an optimized version of the standard real-time driver as distributed by the RTnet open-source project (Kiszka, Wagner et al. 2005) we have implemented a request/reply protocol through which the host can access the CNI and the local configuration parameters of the TISS.

5.2 Experimental Evaluation of Encapsulation

As part of the prototype implementation, we have also performed an early experimental evaluation for validating the encapsulation mechanisms provided by the trusted subsystem. The goal of the experiments is to test the spatial and temporal partitioning in the presence of an arbitrary behavior of faulty hosts. Therefore, the prototype setup contains a micro component with a host performing fault injection, as well as a monitoring device realized by a standard PC. As part of the experiments, the faulty host has systematically iterated through all possible message header values of a request message destined to the TISS.

The monitoring device has been connected to the TTE switch and has used its Ethernet interface in promiscuous mode. The tool WireShark has been used to observe and log the traffic of the entire TTE network. The analysis of the collected logs has yielded the following results:

- ***No discontinuities.*** In the experiments, all sent messages have included sequence numbers. The logs have indicated that no messages have been lost due to the behavior of the faulty lost.
- ***No additional messages.*** Other than the messages configured by the TNA, no messages have been observed on the TTE network.

• ***No effect on temporal properties.*** The temporal properties (i.e., bandwidth, latency, message order) of the TTE network with fault injection by a host have been identical to the temporal properties without fault injection.

In extension to these encouraging results, a more comprehensive validation using fault injection experiments is required as part of future work, e.g., focusing on physical fault injection with radioactive particles.

6. CONCLUSION

The advances of the semiconductor industry are leading to the appearance of billion transistors SoCs, where multiple computers – called *cores* – can be implemented in a single die (Kahle, Day et al. 2005). These developments are opening new alternatives for the design of an integrated architecture for embedded control systems. By assigning each job of a distributed application subsystem (DAS) to a core of a multi-core SoC, the *direct interference* between jobs of different DASes is eliminated without the introduction of a complex partitioned operating system. The *indirect interference* among DASes is eliminated by the provision of a dynamic time-triggered NoC that is controlled by a trusted network authority. This paper has focussed on the error containment properties of the integrated TT-SoC architecture, which in our opinion is superior to the error containment that can be achieved in federated architectures where ECUs are interconnected by a CAN network. The paper contains experimental data on a prototype implementation of the architecture that support our claims.

ACKNOWLEDGEMENTS

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

REFERENCES

Avizienis, A., J. Laprie, et al. (2001). Fundamental concepts of dependability.

Beal, D., E.Bianchi, et al. (2000). "RTAI: Real-Time Application Interface." Linux Journal.

Bosch (1991). CAN Specification, Version 2.0. Stuttgart, Germany, Robert Bosch Gmbh.

Cristian, F. and H. Aghili (1985). Atomic Broadcast: From simple message diffusion to Byzantine agreement. Proc. 15th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-15).

Gray, J. (1986). Why do Computers Stop and What can be done about it? Proc. of the 5th Symp. on Reliability in Distributed Software and Database Systems. Los Angeles, CA, USA.

Hammett, R. (2003). "Flight-critical distributed systems: design considerations [avionics]." IEEEAerospace and Electronic Systems Magazine **18**(6): 30–36.

Hecht, H. (1976). "Fault-Tolerant Software." ACM Computing Survey **8**(4): 391-407.

Heinecke, H., et al. (2004). AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. Proc. of the Convergence Int. Congress & Exposition On Transportation Electronics.

Isermann, R., R. Schwarz, et al. (2002). "Fault-Tolerant Drive-by-Wire Systems." Control Systems Magazine **22**: 64-81.

Jones, C. H., H. Kopetz, et al. (2001). Revised Conceptual Model of DSOS, University of Newcastle upon Tyne, Computer Science Department.

Kahle, J. A., M. N. Day, et al. (2005). "Introduction to the Cell multiprocessor." IBM Journal of Research and Development **49**(4/5): 589–604.

Kiszka, J., B. Wagner, et al. (2005). RTnet – a flexible hard real-time networking framework. Proc. of 10th IEEE Int. Conference on Emerging Technologies and Factory Automation.

Kopetz, H. (1992). Sparse time versus dense time in distributed real-time systems. Proc. of 12th Int. Conference on Distributed Computing Systems. Japan.

Kopetz, H., A. Ademaj, et al. (2005). The Time-Triggered Ethernet (TTE) design. Proc. of 8th IEEE Int. Symposium on Object-oriented Real-time distributed Computing (ISORC).

Kopetz, H. and G. Grünsteidl (1994). "TTP– a protocol for fault-tolerant real-timesystems." Computer **27**(1): 14–23.

Kopetz, H. and R. Obermaisser (2002). "Temporal composability." Computing & Control Engineering Journal **13**: 156–162.

Lamport, L. and R. Shostak (1982). "The Byzantine Generals Problem." ACM Trans. on Programming Languages and Systems **4**(3): 382–401.

Leohold, J. (2005). Automotive Systems Architecture. Architectural Paradigms for Dependable Embedded Systems. Vienna, Austria**:** 545–592.

Normand, E. (1996). "Single Event Upset at Ground Level." IEEE Trans. on Nucl. Science **43**: 2742.

Obermaisser, R., et al. (2006). "DECOS: An Integrated Time-Triggered Architecture." journal of the Austrian professional institution for electrical and information engineering **3**: 83–95.

OMG (2002). Smart Transducers Interface Specification, Object Management Group.

Pauli, B., et al. (1998). "Reliability of electronic components and control units in motor vehicle applications." VDI Berichte 1415, Electronic Systems for Vehicles: 1009–1024.

Saib, S. H. (1987). Executable Assertions - An Aid to Reliable Software. Proc. of Asilomar Conference on Circuits Systems and Computers.

Suri, N., C. J. Walter, et al. (1995). Advances in Ultra-Dependable Distributed Systems, Chapter 1, IEEE Computer Soc. Press.

Wilkinson, C. (2005). "IMA aircraft improvements." Aerospace and Electronic Systems Magazine, IEEE **20**(9): 11–17.

GENERIC ARCHITECTURE DESIGNED FOR BIOMEDICAL EMBEDDED SYSTEMS

L. Sousa, [1] M. Piedade, [1] J. Germano, [1] T. Almeida, [1] P. Lopes[1],
F. Cardoso, [2] and P. Freitas [2]

[1]*INESC-ID/IST, TULisbon*
1000-029 Lisboa
Portugal
{las,msp,jahg,tmma,paulo.c.lopes}@inesc-id.pt

[2]*INESC MN/IST, TULisbon*
1000-029 Lisboa
Portugal
{fcardoso,pfreitas}@inesc-mn.pt

Abstract: Embedded Systems assume an increasing importance in biomedical applications such as clinical analysis and patient monitoring. The lack of generic architectures make the design of this type of autonomous embedded systems a cumbersome and expensive task. This paper proposes a generic architecture for developing biomedical embedded systems, considering both the hardware and the software layers. A prototype of one of these systems for biomolecular recognition, based on magnetoresistive sensors to detect magnetic markers, was already implemented by off-the-shelf components usage. Experimental results show the effectiveness of the proposed architecture and the advantage of its application to develop distributed biomedical embedded systems.

Keywords: Embedded systems, biomedical applications, computing architectures, autonomous communication systems

1. INTRODUCTION

In the last few years there has been a crescent interest on embedded systems for biomedical applications, increasing the demand on computing and communication while, at the same time, maintaining the necessity of a portable and autonomous system. Applications such as biochemical operations for clinical analysis (e.g, glucose/lactate analysis), DNA analysis and proteomics analysis for clinical diagnostics [Piedade et al., 2006] and real-time pervasive patient monitoring [Jovanov et al., 2001] [Halteren et al., 2004] are typical examples where high computing and communication requirements must be effective.

Sousa, L., Piedade, M., Germano, J., Almeida, T., Lopes, P., Cardoso, F., Freitas, P, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 353–362.

Portability and computing power are requirements that lead to integration of wireless devices on embedded systems in order to communicate with general purpose computing systems and also with the setup of distributed computing platforms based on all these computing engines.

Actual embedded systems for biomedical applications have to be equipped with low power communication systems and, on the other hand, have to be easily integrated with more general distributed computing platforms where reliability and security issues have to be considered, both at the computation and the communication levels. The enormous diversity of sensors and medical apparatus demand the development of general computation/communication architectures for deploying distributed embedded systems that may cover a wide range of applications and environments.

This paper proposes a communication architecture for implementing a distributed platform that supports autonomous embedded systems for medical applications. The considered architecture includes both the hardware and software components and allows the development of autonomous but collaborative embedded systems through actual technologies usage. Moreover, the paper presents the application of the proposed architecture for developing a handheld microsystem for biomolecular recognition, based on an integrated magnetoresistive biochip. Experimental results obtained with a system prototype show that the proposed architecture is flexible and may be directly applied in designing embedded systems for several different applications and that it can be easily adjusted to fulfil all particular requirements of each biomedical experience.

2. PROPOSED ARCHITECTURE

Figure 1 presents the block diagram of the hardware component of the proposed architecture. In the core of the Autonomous Communication Platform (ACP) there is a communication manager, which is responsible for communicating data and commands from and to a local Acquisition/Processing Platform (APP). Each pair of these two platforms are tightly coupled through a Serial Peripheral Interface (SPI) with a simple protocol but with large a bandwidth. Together they compose an embedded system which communicates with more general computing devices, here designated as master devices, such as a laptop or a Personal Digital Assistant (PDA), through communication modules. These communication modules implement all the necessary protocols and provide electrical interface for serial yet standard protocols, wire-based or wireless, such as the Universal Serial Bus (USB), Bluetooth or Wi-Fi.

As depicted in fig. 1, multiple portable communication/processing platforms may be connected to a single master by using the capacity of the communication standards to set up small networks, e.g., the Bluetooth that uses a radio-

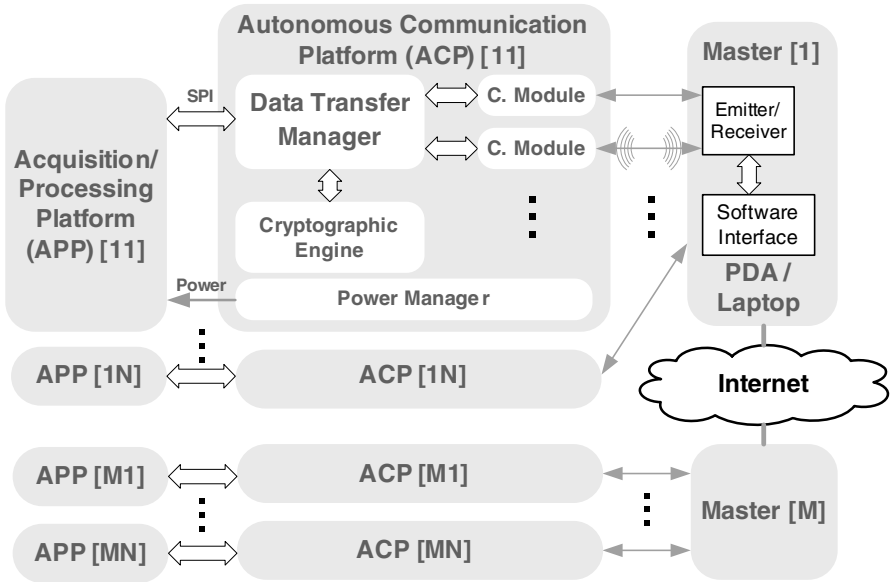


Figure 1. Block diagram of an implementation of the proposed architecture.

based link to establish a connection whenever devices come within 10 meters of each other. Furthermore, masters act themselves as a second communication layer by directly using the IEEE 802.11 standard for wireless Local Area Networks (LANs). We propose the usage of the HyperText Transfer Protocol (HTTP) and WebServices in order to develop a distributed environment in which different masters can be relatively far away and connected by a Wide Area Network (WAN). For security reasons it is advisable to adopt an implementation of the HTTP on the top of Secure Sockets Layer (SSL) or Transport Layer Security (TLS) leading to HTTPS.

Two additional important blocks are present in the ACP: the power manager and the cryptographic engine. Since the platform is autonomous it has to be equipped with a battery. The power manager is responsible for monitoring the state of this battery and for providing its recharge whenever it is necessary. When using some buses, such as the USB, the controller is also able to provide electrical power to other devices. If such is the case, this electric energy can also be supplied to the acquisition/processing platform, which usually also has to be autonomous.

The cryptographic engine present in the ACP assures privacy and integrity during data transfer to the master. In the particular case of this application, a public-key or a symmetric cryptosystem can be applied. Examples of cryptographic algorithms that may be applied are the Elliptic Curve Cryptography (ECC) and the Advanced Encryption Standard (AES) [Schneier, 1995], which

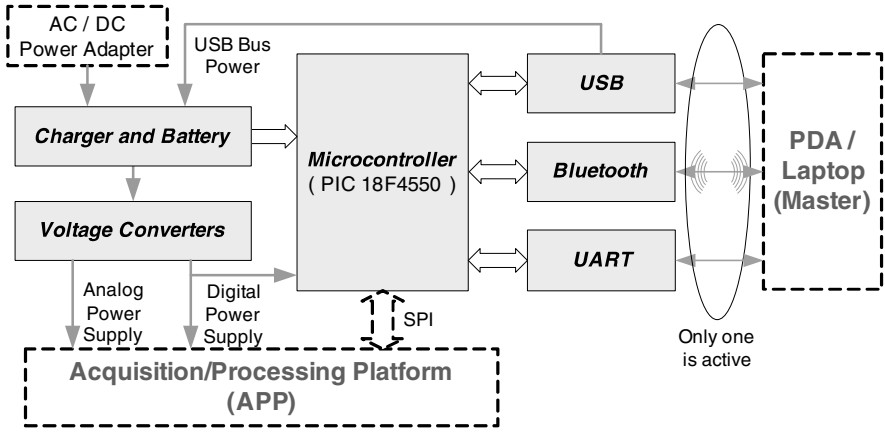


Figure 2. Block diagram of the hardware component of the proposed architecture.

require small key sizes and allow high strength per bit. For example for the ECC, there are also available some optimized implementations for portable and autonomous devices with modest computational capacity [Weimerskirch et al., 2001]. Other encryption algorithms that require less computation resources can also be considered for the communication platform if extremely low power operation is required.

Figure 2 presents a block diagram of an actual implementation of the autonomous and portable communication platform. At the core of the platform there is an off-the-shelf microcontroller (PIC) that integrates a USB stack and transceiver [Microchip, 2004]. This PIC manages all communications with the ACP through a SPI simple protocol with a transfer rate up to 30 Mbps. This interface is used to transfer data and commands between the acquisition and processing platforms. The PIC also provides the universal asynchronous receiver/transmitter protocol which can be directly used to connect the platform or may be used as an interface to feed data to a Bluetooth communication module (e.g. [Bluegiga Tech., 2006]). These different communication modules allow communicating at different rates and distances, for example 12 Mbps for USB while 2 Mbps are achieved to communicate up to 10 m with the Bluetooth. As it is depicted in the diagram, the USB can also supply electric energy to both the communication and the acquisition/processing platforms.

Communication with the APP is performed using a simple protocol that only uses two types of data packets. First, using a fixed length packet, a command is send followed by the number of data values to be transmitted. Then the receiver interprets the command and send an acknowledge signal. Finally, using a variable length packet, the sender transmits the specified number of data values. Commands can also be divided into two classes: configuration and ac-

quisition. Although, the packet structure is the same, the acquisition command only generates an additional variable length packet sent by APP. The communication between the master and the ACP also uses a similar protocol, but the acknowledge signal is implemented in a different manner. The PIC is also responsible to implement the encryption algorithm when secure data transfer to the master is required. Whenever the master enables this feature data packets sent by the APP are encrypted before being sent to the master.

The power manager block is composed by battery charge circuits with voltage sensing that can draw energy provided from the USB master or, alternatively, it gets the energy from an external power supplier. The sensing circuit is used for applying low power consumption techniques at the software level and also by dynamically adjusting the microcontroller operating frequency. Supply voltages required by the platform components are obtained through high efficient switched voltage converters usage.

The software of the communication platform was mostly written in C, but some of the critical parts were coded in assembly language. The encryption engine is an optional software module that encrypts messages before sending them to the master. This is one of the modules that can be switched off by the power manager. To develop the software needed by the masters, the Microsoft operating system and tools were adopted. Classes were programmed in C++ for decryption, communication and general user interfacing, by using the Visual Studio environment. Communication between masters is made by exchanging requests or replies based on the Simple Object Access Protocol (SOAP) via WebServices. WebServices provide a request acceptance and a reply service by using Extensible Markup Language (XML) based files. When a master needs to send a request, request acceptance services only have to write the request XML file to a given directory in the server machine and to return a request ID. Reply services simply return the content of a XML file related to the request ID, if it exists on a reply directory of the server machine. Therefore, masters just have to exchange request/reply data.

3. ARCHITECTURE APPLICATION TO AN EMBEDDED MICROSYSTEM FOR BIOLOGICAL ANALYSIS

The architecture described in the previous section was applied to develop an embedded hand-held microsystem for biomolecular recognition assays, using target biomolecules marked with magnetic particles.

Labelled targets are recognized by biomolecular probes immobilized on the surface of the chip over sensing sites and the markers fringe magnetic fields are then detected by magnetic sensors [Piedade et al., 2006]. The main advantage of this type of microsystem is the possibility to directly detect biomolecular

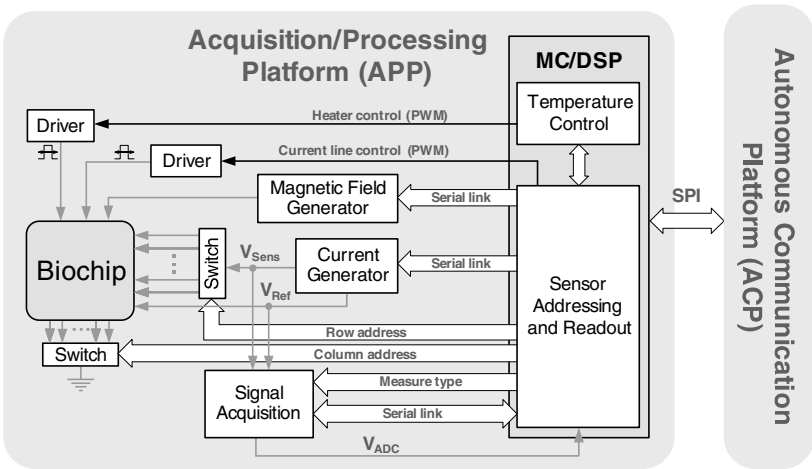


Figure 3. Block diagram of the acquisition/processing platform of the microsystem for biological analysis.

recognition (e.g. DNA hybridization) by reading the magnetic field created by the markers using a sensor located below each probe site. The action of taking an electrical measurement, instead of an optical one, reduces considerably the readout system complexity and increases sensitivity.

Figure 3 presents the Acquisition/Processing Platform of the microsystem for biological analysis, which is connected to the ACP through a SPI. It generates all electric signals to drive the sensor array and to individually address and readout signals provided by each of the sensors. Moreover, it is also in charge of individually measuring and controlling the temperature in subsections of the biochip, by using both the biochip Current Lines (CL)s and by taking advantage of the Thin Film Diode (TFD) voltage-temperature characteristic [Piedade et al., 2006].

Sensor addressing is based on a commutating matrix of integrated TFDs, each one acting as a temperature sensor and as a switch in series with a Magnetoresistive Tunnel Junction (MTJ) which is the magnetoresistive sensor. The microcontroller provides row/column addresses to sensor reading and define the drive current needed through a digital-to-analog converter (DAC). This allows the usage of a single DAC and a single instrumentation amplifier to drive and to read all the sensors in the matricial array. These are the only analog circuits, since control and signal processing are performed by digital processors associated to a 1 Mbit memory for storing all acquired and processed data.

The TFDs, acting as temperature sensors, are calibrated by programming the digital processor to generate current pulses modulated in width (PWM). The calibration is performed in DC, by amplifying a voltage at the terminals of the

serial circuit in each sensing site. This calibration phase is performed at setup time in order to experimentally extract TFDs parameters that allows voltage-temperature characterization. Calibration tables are filled for each sensor with absolute and differential voltages measured using reference sensors available on the biochip. To measure MTJs resistance variation due to magnetic field variation, an AC excitation is performed, using an external magnetic field generated by a coil placed below the biochip. The generation of this magnetic field is digitally controlled. This AC analysis allows the measurement of small relative resistance variations (less than 1%) by using differential mode of amplification. The necessary reference signal can be generated by a microcontroller or registered from the sensors themselves in specific operating conditions.

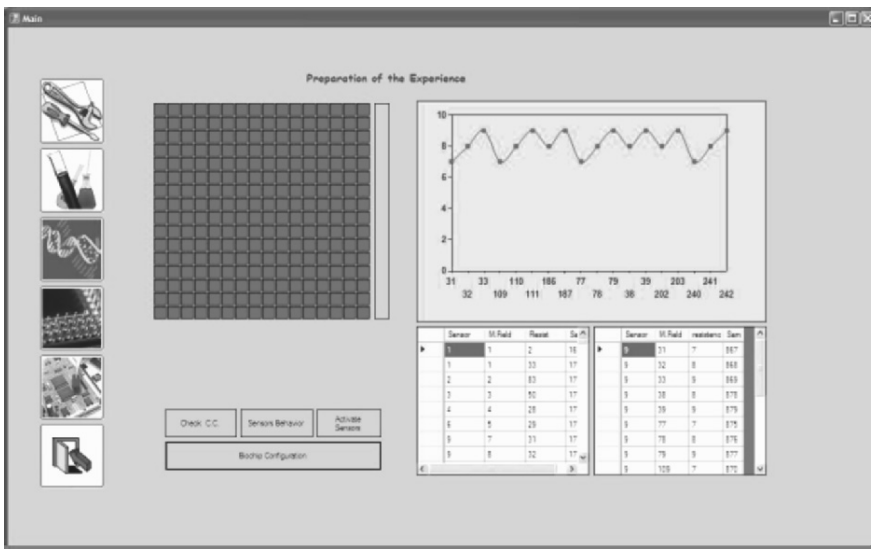


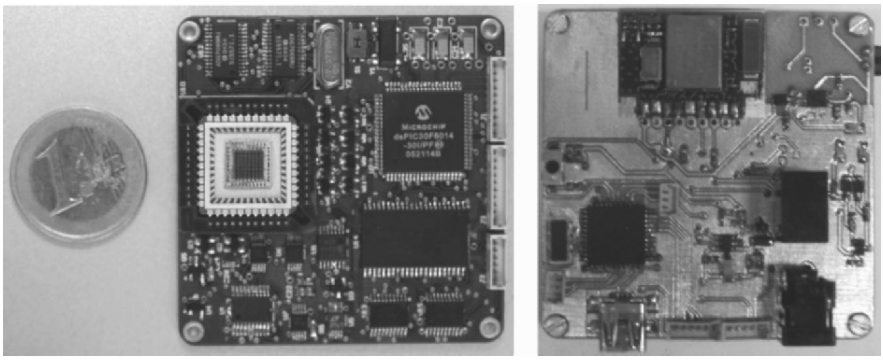
Figure 4. Snapshot of the menu for preparing the experience taken directly from the laptop display.

For this particular case, a Pocket Loox 720 with an Intel XScale PXA 272 520 MHz processor, 128 MB RAM memory, Bluetooth 1.2 and USB 1.1 host capabilities was used. The master may also be hosted in a laptop computer with minimal changes on the software. This master device can perform all the necessary data analysis or it can send the data to be further processed in other master devices. One of the menus of the graphical user interface provided by the master is presented in fig. 4. This particular menu allows the configuration of the experience to be performed, the definition of the biochip geometry, in this particular case a 16×16 matrix, the activation/deactivation of some of the sensors and their main characteristics extraction. Other more detailed sub-

menus are available in order to define the type of excitation to be applied to the sensors and to choose which measures have to be collected and registered.

4. IMPLEMENTED PROTOTYPE AND EXPERIMENTAL RESULTS

The described architecture of the communication platform was implemented on a conventional two layer board PCB with room for the Bluetooth module. Because the biological analysis platform measures signals with a slow variation, the use of a Bluetooth module which provides a lower bit rate but consumes less power was considered. The achieved board has a square shape with an area of about 32 cm^2, which is smaller than a credit card. The size of the communication board is the same of the acquisition board, allowing the boards to be stacked and thereby making the system more compact. The implemented prototype of the communication platform and the acquisition and processing board are depicted in 5.



(a) Biological analysis platform. **(b)** Communication platform.

Figure 5. Prototypes of the embedded microsytem.

According to the experimental results obtained for ACP power consumption it exhibits an autonomy of about 30 hours, when the platform is continuously operating at full speed and a battery of about 700 mAh is used. This autonomy can be significantly increased by the power manager if the operating frequency is properly adjusted. For the envisaged application the acquisition sampling rate is low and, in order to save power, the Bluetooth transfer rate is set to 9600 kbit/s. Due to communication overheads the effective data transfer rate drops to 7000 kbit/s, which still fulfils biological analysis system requirements. Some results were also achieved for the cryptographic engine. In this prototype it is used the AES, with 16-byte key size applied to 16-byte block of data,

as there are some optimized implementations available for the target microcontroller [Microchip, 2005]. Since this encoding method only requires 367 instruction cycles per byte, it still can be used to secure data in real-time data acquisition.

The embedded system prototype was tested using a solution of 2.3×10^9 particles/ml with 1.5 μm diameter magnetic nanoparticles. A 5 μA DC current was driven by the DAC through a 10 kΩ MTJ. The voltage signal was measured by an ADC at a sample rate of 6 Hz after passing through a suitable anti-aliasing filter and the measurement time was about 80 seconds. The solution was dropped on the sensor after about 20 seconds and after more 30 seconds the sensor was washed with distilled water. The acquired data was registered on the PDA and sent to a desktop computer using the SOAP. This data is graphically represented as a web page in fig. 6, after the removal of a 47 mV DC signal. This web page was generated using php 5.0 hosted in an Apache 2.0 web server, in order to visualize the received XML data file. The graphic is drawn through the use of the JpGraph object-oriented graph creating library in order to generate a png file that can be interpreted by a web browser.

These experimental results clearly shows a 190 μV signal due to the presence of nanoparticles, demonstrating that que the developed embedded system can be used for particle detection.

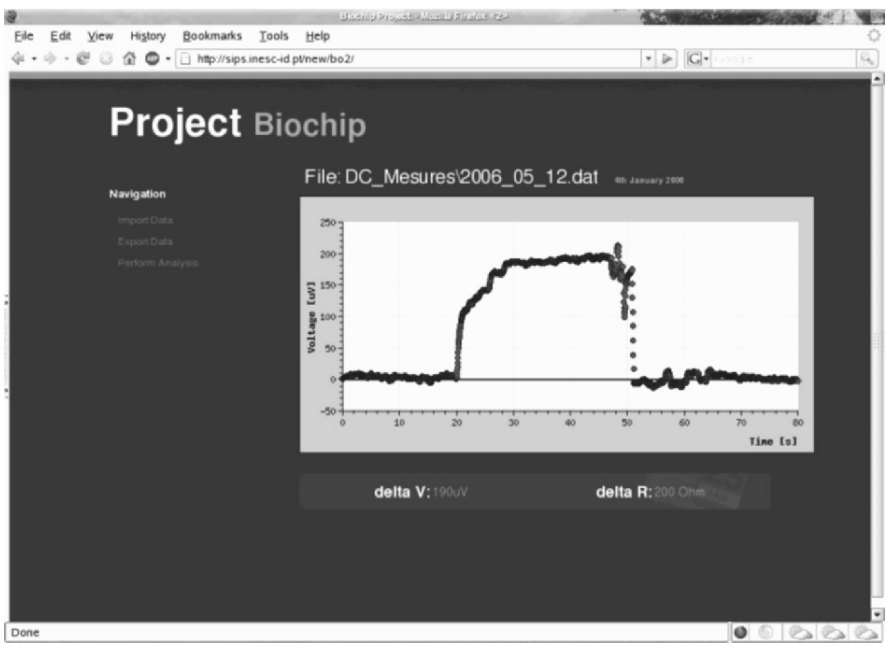


Figure 6. Snapshot of the web-based XML data visualization.

5. CONCLUSIONS

This paper presented a new generic and modular architecture for designing biomedical embedded systems. Since biomedical programmable systems are considered, both hardware and software components are important. The developed platform is adaptable to the specific requirements of a given acquisition platform. The effectiveness of the proposed architecture was shown by implementing a prototype of a biomolecular recognition system. This embedded system is based on magnetoresistive sensors to detect magnetic markers and it was implemented with off-the-shelf components. The prototype includes all the devices required to communicate with a master system and provides security mechanisms based on data encryption. The achieved data transfer rate of 7000 kbit/s is suitable for the acquisition platform but can be increased if required. Finally, the low power consumption of this prototype allows its autonomous full speed operation for more than 30 hours. These experimental results show the interest of the proposed architecture to develop distributed biomedical embedded systems.

REFERENCES

[Bluegiga Tech., 2006] Bluegiga Tech. (2006). WT12 Bluetooth Module. Version 2.3.

[Halteren et al., 2004] Halteren, A., Bults, R., Wac, K., Konstantas, D., Widya, I., Dokovski, N., Koprinkov, G., Jones, V., and Herzog, R. (2004). Mobile patient monitoring: The mobihealth system. *The Journal on Information Technology in Healthcare*, 2(5):365–373.

[Jovanov et al., 2001] Jovanov, E., Raskovic, D., Price, J, Chapman, J, Moore, A, and Krishnamurthy, A (2001). Patient monitoring using personal area networks of wireless intelligent sensors. *Biomed Sci Instrum*, 37:373–378.

[Microchip, 2004] Microchip (2004). PIC18F2455/2550/4455/4550 Data Sheet. ref: DS39632C.

[Microchip, 2005] Microchip (2005). Data Encryption Routines for the PIC18. ref: DS00953A.

[Piedade et al., 2006] Piedade, M., Sousa, L., Almeida, T., Germano, J., Costa, B., Lemos, J., Freitas, P., Ferreira, H., and Cardoso, F. (2006). A new hand-held microsystem architecture for biological analysis. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 53(11):2384–2395.

[Schneier, 1995] Schneier, Bruce (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 2nd edition.

[Weimerskirch et al., 2001] Weimerskirch, A., Paar, C., and Shantz, S. (2001). Elliptic curve cryptography on a palm os device. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy*, volume LNCS, 2119, pages 502 –513.

A SMALL HIGH PERFORMANCE MICROPROCESSOR CORE SIRIUS FOR EMBEDDED LOW POWER DESIGNS, DEMONSTRATED IN A MEDICAL MASS APPLICATION OF AN ELECTRONIC PILL (EPILLE®)

Dirk Jansen, Nidal Fawaz, Daniel Bau, Marc Durrenberger
Institut für Angewandte Forschung, Hochschule Offenburg
Badstrasse 24, 77652 Offenburg, Germany
email: d.jansen@fh-offenburg.de

Abstract: A new, small, and optimized for low power processor core named SIRIUS has been developed, simulated, synthesized to a netlist and verified. From this netlist, containing only primitives like gates and flip-flops, a mapping to an ASIC - or FPGA technology can easily be done with existing synthesizer tools, allowing very complex SOC designs with several blocks. Emulation via FPGA can be done on already simple setups and cheap hardware because of the small core size. The performance is estimated 50 MIPS on Cyclone II FPGA and about 100 MIPS on a 0.35 CMOS 5M2P technology with 4197 primitives used for the core, including a 16 x 16 multiplier. An example design of the ASIC for an electronic ePille device currently in development is shown.

Key words: Processor core, System-on-Chip, embedded systems, ASIC, electronic capsule, and medical application

1. INTRODUCTION

Small processor soft cores, well suited to mobile applications with low power consumption, made from few logic cells, and easy to program are still not easily available on the market, although there are many offers of IP cores

Jansen, D., Fawaz, N., Bau, D., Durrenberger, M, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 363–372.

from different suppliers. Soft cores shall be defined here either as a HDL description in VHDL or Verilog, or as a RTL- net list of primitive instances as there are logic gates, flip-flops and memory blocks, available in every library for ASIC/FPGA design or other common user reconfigurable technology.

The open cores website [1] offers several clone processor cores based on the well known legacy 8048/8051 8bit concept, some few on Z80 and 6800 basis and some 16bit and 32bit high level cores like MIPS. A remarkable high level core from Gaisler Research ltd. is LEON 3 [2], basically a SPARC 32 with many features and extensions. There are several IP companies in the market which are licensing processor cores, most important ARM ltd. with its ARM 7 thumb family and ARM 9xx, which are both high performance 32 bit cores. Other companies are ARC ltd, with a scalable, reconfigurable core design, and the free available OpenRISC_1200 with several implementations. Besides of the license costs, full performance is only available after a more or less expensive porting process to a target technology.

There will be more and more devices containing programmable architectures (FPGA). Now with the availability of low power devices (IGLOO-series, MAX II-series, and PolarPro-series), designers will use soft processor cores with these devices for battery powered designs too, gaining a bigger slice from the electronic IC market.

Here we report on the application of a soft processor core used in a (potential) mass application for an ePille®, an electronically controlled disguisable medical drug delivery device, which may replace in part the actual chemical drug release mechanisms used today. First dominant application will be non bloody delivery of insulin to patients with diabetes mellitus disease.

The paper is organized in the following way: Chapter 1 describes the main specification and goals of the design. Chapter 2 shows instruction set architecture and block schematics. Chapter 3 gives synthesis results and performance data. Chapter 4 reports how a concrete SOC design of an ePille was made with this soft core. Chapter 5 gives achieved performance results.

2. DESIGN GOALS

The SIRIUS core (acronym for Small Imprint RISC for Ubiquitous Systems) stands in performance somewhere between the very successful architectures of the ATMEL AVR (ATmega 8bit) , the TI MSP 430, the PicoBlaze - and well below the ARM 7/9– class of 32bit machines, the LEON (SPARC), Motorola 68xxx and other 32bit architectures (NIOS II,

MicroBlaze). Although benchmarking still has to be done, the goals are easy to summarize:

- Load-Store architecture with 16bit external data bus.
- RISC architecture with 3 stage pipeline and 1 Instruction/clock cycle and an average of 0.8 MIPS/MHz performance.
- 16bit/32bit internal bus structure with 32bit ALU and 16x16 MPY,
- Orthogonal register set of 16 registers, 12 x 16bit, 4 x 32bit, the 16bit universal registers may be combined to double registers and handled as 6 x32bit registers.
- Instruction pointer 32bit and stack pointer 32bit are part of the register set.
- Stack oriented architecture with unlimited nesting levels.
- Pointers 16 bit as well as 32bit supported.
- Multiplex bus system, separated 8bit IO bus and fast memory bus, all IO is connected via a separate 8bit bus with own address space.
- Address space 64k or 4G depending on pointer addressing,
- Vectored hardware interrupt and software interrupt (exception)
- Compact 16 bit instruction format with only three modes.
- Instruction set architecture with 56 instructions optimized for compact C-Compilation.
- Net list version made from gate primitives, able to be mapped on every existing technology without using macros.

In the basic version, a J. v. Neumann architecture with common program/data memory and 16bit bus is used. Fig 1 shows a block diagram of the core. One of the main ideas was to create a scalable architecture, which can be fitted to every application. SIRIUS can be used for basic one chip designs, containing everything, as well as for Linux based designs with addressing capability of hundreds of Megabytes, using external DRAM. The design is further optimized for setting up a virtual machine for JAVA processing. Any NOP instructions and any limitations in using instructions are avoided, allowing still assembler programming where it makes sense.

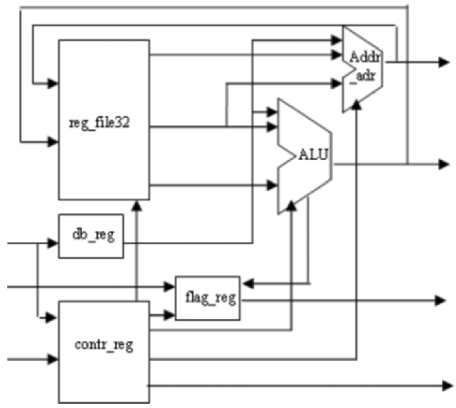


Figure 1. Block diagram of the SIRIUS soft core

The goal of the SIRIUS development is a core with similar or improved features than the existing 16bit class of processors, but with better programmability and a high efficient C-Compiler for comfortable programming. Most of all instructions run in 1 cycle with a 3 phase pipeline structure. There is a 32bit infrastructure for internal data handling and ALU processing. As a result, 32bit performance is not worse than 16bit performance, beside loads and stores need additional cycles. Instructions for 32bit and 16bit are using the same mnemonics, so there are only few codes to remember.

3. INSTRUCTION SET ARCHITECTURE

The programmer's model is shown in Fig. 2. The instruction set is a nearly direct replacement of the DAG operators, defined by Fraser and Hanson [3] with their re-targetable Little C – Compiler (LCC) [4]. ISA is a result of optimization for C-Compilation. So the compiler was defined first and the instruction set then deviated. Fig 3 shows the code space for a 6bit operation code. From 64 available different codes, 56 are actually used for instructions. The instruction set is near minimum, but with the immediate instructions added. These instructions use a constant as one argument, which makes these instructions 32/48 bit long. See Fig 4 for the instruction formats. Load/Store instructions are always based on a pointer, which is hold in a register, normally called indexed addressing. This pointer can be 16bit or 32bit, depending on attr0, the data can be 16bit or 32bit long, depending on attr1. Calls are again based on pointers, 16bit or 32bit, the return address pushed on the stack is always 32bit, allowing a large memory model used. Both addressing modes can be mixed, allowing flexible memory assignment.

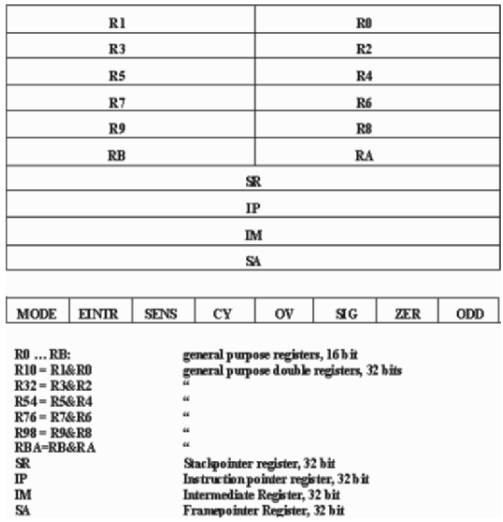


Figure 2. Programmer's Model

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	SAL	SAR	SLL	SLR	CLU	CUL	CLI	CIL	MPU	ADD	SUB	CMP	ANA	ORA	XRA
1	MOV	SALc	SARc	SLLc	SLRc	INV	INC	DEC	LXI	LDI	ADI	SBI	CPI	ANI	ORI	XRI
2	POP	PSH	PIN	POT	JMP	CAL	RET	SWI	BOV	MPS	--	--	--	--	--	--
3	LDB	STB	LDX	STX	BCY	BEQ	BGT	BGE	BNE	ITR	PSF	PPF	RST	ENI	DIS	HLT

Figure 3. Code space for 6bit operation code, RST is reset and ITR is interrupt, not usable as instructions, 6 codes are still free for extension

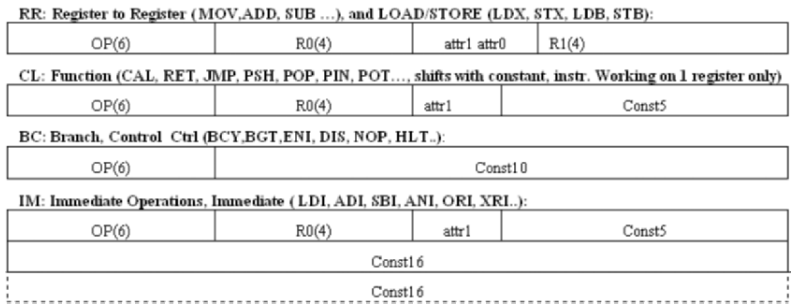


Figure 4. Instruction Formats

Branches are always relative branches in relation to the instruction pointer; the branch-offset is limited by the size of the constant to +/- 1023

bytes or 511 Instructions, normally sufficient. Branches with larger offsets have to be replaced by Jumps, which may have every size, which is done by the assembler. All these instructions are of 16bit format.

The instructions are pipelined during execution with the basic 3 stage pipeline scheme

FETCH – DECODE – EXECUTE.

Write back is included in EXEC for register to register instructions. The flip-flop based register set structure allows to read a register, to process the data in the ALU, and to write into the register in only one clock cycle. The only exception is the MPY instruction, which is pipelined itself and uses 2 EXEC cycles, and the load/store instructions, which need 3 cycles, because of the 16bit limited data bus size and the J. v. Neumann architecture. This gives a better balancing of delays and allows higher clock frequencies for register to register processing. Fig 5 shows an example for pipelining. In general, the pipeline depth is adapted to the delay of a memory read, so memory acquisition delay of the target technology mainly defines the maximal frequency of the core.

AdMM	Addr0	Addr1	Addr2	Addr3	Addr4	Addr5	Addr6	Adrdat	Addr6	Addr8	Addr9	Adrdat	Addr9		
db_in		Instr0	Instr1	Instr2	Data2	Instr4	Instr5	Instr6	Data5	Instr6	Instr8	Instr9	Data8	Instr9	
Nmoncs			MOV	ADD	LDI1	LDI2	ADD	LDX1	LDX2	LDX5	SUB	STX	STX	STX	MOV
ireg			Instr0	Instr1	Instr2	Instr2	Instr4	Instr5	Instr5	Instr5	Instr6	Instr 8	Instr8	Instr8	Instr9
db_reg			Instr0	Instr1	Instr2	Data2	Instr4	Instr6	Instr6	Data5	Instr6	Instr8	Instr9	Data8	
regs				Res0	Res1		Res2	Res4			Res5	Res6			

Figure 5. Pipelining of a program sequence, showing 1 cycle MOV, ADD, SUB, 3 cycle LDX, STX and 2 cycle LDI

4. SYNTHESIS RESULTS AND PERFORMANCE

The core was coded in VHDL and synthesized to a VERILOG net list, using the design compiler DC of Synopsys Inc. As a target technology, a selected and generalized 0.35µm technology library was used. This implicates the following advantages as seen in Fig 6:

- Timing is optimized like for an ASIC design, but without concrete mapping to an existing technology.

- The netlist can easily be remapped with DC to any CMOS target technology without significant changes and risks, e.g. to 0.35µm AMI, or 0.13µm UMC or other technologies.
- A complete timing based simulation (SDF File from Synopsys DC) with MODELSIM is possible, same as a static timing analysis.
- The netlist can be loaded to any FPGA design IDE like ALTERA-QUARTUS together with a basic (behavior)-library of the primitives used in the netlist. This allows controlling the detailed behavior via emulation. The tool normally remaps and optimizes the netlist to the used FPGA target technology. The only disadvantage is that available special cells like multipliers in the FPGA are not used; because they are dissolved to primitives (This can be overcome with a special version of the net list for FPGA synthesis).

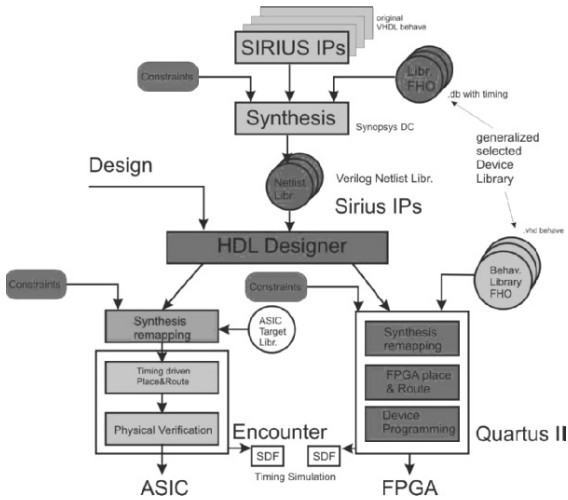


Figure 6. Netlist based IP development and dissemination

The only disadvantage in the above method is the higher effort in simulation because of the fine grain structure of the netlist, which leads to longer simulation times (not really significant on fast computers). For larger software simulations, a software instruction set simulator or the emulation on the FPGA is more effective. This is again true for driver development for new IO-blocks.

Low power consumption is based on the following methods: Only registers are clocked, which are enabled and have to be written. Circuitry, which is not used, is disabled (e.g. the multiplier is logically disconnected when not used). All busses are made in form of multiplexer architectures, no tristates used. Using a low power target library.

In normal operation mode, only a small part of the logic is active. In addition, power can be optimized by the DC, using power constrains. Real consumption can only be discussed after defining the target process, the supply voltage, clock frequency and the temperature range [5]. The HLD mode, which is entered by the hold instruction and can only be left by reset or an interrupt, keeps only very few gates active and is a natural low power mode. With the achieved low number of cells for the core, static power is in the below microampere range, which is important for power down or sleep modes.

5. PUTTING THE SIRIUS CORE IN AN EPILL APPLICATION

The first project which shall demonstrate the performance of the core is a medical application, where a disguisable drug container (ePille®) Fig 7 shall release its medicament by remote commands via a near field communication (NFC) interface.

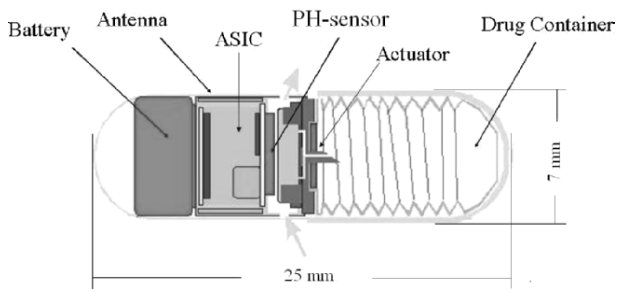


Figure 7. ePille concept of an electronic drug delivering device

Due to some additional requirements, sensor processing etc, which shall not further discussed here, a complex controller ASIC is set up, see Fig 8. The core is the heart of the circuit; all peripheries are connected via the peripheral 8bit bus. There is a parallel port as well as a serial IO, SPI and I²C Interface, a modem block for DQPSK communication [6] and a NFC block for ISO 14443a RFID communication [7], depending on the requirements. Watchdog and timer blocks are classic. Power down mode with clock switching allows together with PLL based main clock oscillator extreme low power consumption. Event handling and wake up is made by a special programmable wake up manager WUM.

Memory in this prototype is made of SRAM, which takes a lot of silicon area, much more than all other circuitry (see Fig 9). It will be replaced in a mass product by ROM or even Flash-ROM which both is much smaller. Booting is done by a small boot-ROM, synthesized from gates, containing a loader, which loads the actual software via the SPI – Interface from an external serial flash memory, which can store additional software and collected data during the usage process. Some of the above mentioned blocks may be spared in a production type of the chip. Fig 9 shows a first routing of the chip, giving an impression on the small size of the SIRIUS core. The routing does not contain all of the analogue blocks, which are partly still under development. The ASIC is designed in 0.35µm CMOS from AMI with 5 metal and 2 poly-layers and has an area of about 11 mm^2, mostly SRAM. The SIRIUS core alone is below 1 mm^2 in size.

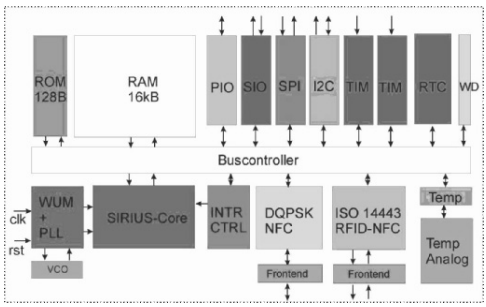


Figure 8. Block diagram of the ePille-controller with SIRIUS processor core

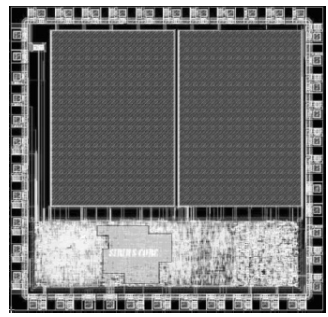


Figure 9. Prototype ePille- SOC-ASIC layout in 0.35 AMI CMOS technology, (not complete), Size is 11mm^2

6. RESULTS

A new, small and for low power optimized processor core SIRIUS has been developed, simulated, synthesized to a netlist and verified via FPGA emulation. The basic concept was already presented in 2005 [8]. Table 1 summarizes the achieved results in synthesis for a 0.35µm AMI library as well as for the low entry FPGA Cyclone II device. Table 2 shows estimated performance from the actual available data, measured results will be presented on the conference.

Table 1. Results of synthesizing for 0.35 ASIC Library and for ALTERA Cyclone II

Block	Prim. (ASIC)	LE (FPGA)
SIRIUS core	4197	2870
ROM	392	106
Bus controller	92	335
PIO	52	20

Block	Prim. (ASIC)	LE (FPGA)
SIO	388	132
SPI	241	123
I2C	343	191
Watchdog	102	50
Intr. Contrl.	773	346
ISO-14443 Interf.	1407	617
QPSK Interface	3186	1252
Timer	214	88

Table 2. Estimated performance data of a SIRIUS based SOC

Performance	Value	Comment
MIPS average/MHz	0.8	Depending on J. v. Neumann architecture and 16b ext. Bus
Max clock frequency (ASIC)	120 MHz	For 0.35 CMOS technology and 3.3 V supply
Max clock frequency FPGA	60 MHz	For Altera Cyclone II and optimized placement
Power in HOLD mode	<20uW	Main clock running, all periphery shut off, ASIC
Power in P D mode	<3 µW	Only RTC running, main clock off
Code density	16bit	High, All instructions 16 bit, near to C-statements
Architecture	RISC	J.v. Neumann, pipelined execution, 3 stage pipeline
Address space	64k / 4G	Depending on pointer size used, both hardware supported
OS feasible	LINUX	Together with external DRAM and large memory model
Registers	12 + 4	16 bit registers can be combined to 32 bit registers
ALU	32 bit	Instructions use 16/32 bit ALU
MPY	16 x 16	Signed/unsigned Multiply in Hardware, 2 cycles
Interrupt	128	Vectored Interrupt with interrupt controller, periphery unit
Instruction set	56	Instructions mostly identical for 16 bit and for 32 bits
High level Language support	LCC	C-Compiler and IDE available, JVM

REFERENCES

[1] Opencores: http://www.opencores.org/
[2] LEON3: http://www.gaisler.com/
[3] Fraser, Christoph and Handson, David: A Retargetable C Compiler: Design and Implementation. Addison Wesley P.C., Redwood City, Ca ISBN 0-8053-1670-1, 1995
[4] LCC Compiler: http://www.cs.princeton.edu/software/lcc/
[5] Jansen, Dirk ed., et. alt.: Electronic Design Automation Handbook, Verlag Kluwer, NL, 2003.
[6] Jansen, Dirk, Fawaz, N.: DQPSK Modulator for Inductive Data Transmission, MPC-Workshop, Reutlingen, June 2002.
[7] Jansen, Dirk, Baier, F.: Induktive bidirektionale Schnittstelle ähnlich ISO/IEC 14443-A, MPC-Workshop, Reutlingen, June 2002.
[8] Jansen, Dirk: Systematic Design of a Small Processor Core with C-Capability for SOC Designs; Presentation at the CECS, University of California, Irvine on July 9, 2005.

UTILIZING RECONFIGURABLE HARDWARE TO OPTIMIZE WORKFLOWS IN NETWORKED NODES

Deploying partial reconfiguration to optimize load and dependability in constrained networks

Dominik Murr[1], Felix Mühlbauer[1], Falko Dressler[2], Christophe Bobda[1]

[1] *Self-Organizing Embedded Systems Group*
Department of Computer Science
Kaiserslautern University of Technology, Germany
Dominik@Murr.com, {Muehlbauer,Bobda}@informatik.uni-kl.de

[2] *Computer Networks and Communication Systems*
Department of Computer Science
Friedrich-Alexander University Erlangen-Nuremberg, Germany
Dressler@informatik.uni-erlangen.de

Abstract: This work investigates the use of reconfigurable devices as computing platform for self-organizing embedded systems. Those usually consist of a set of distributed, autonomous nodes interacting with each other in order to solve a given problem. Several aspects of hardware-software co-design as well as partial reconfiguration are presented in order to enforce adaptivity of a node. One targeted application field for this kind of system are sensor networks in which reconfigurable devices, in this case FPGAs, can be used as computation nodes to provide services that require more computation power. To manage the available hardware resources as a whole we suggest a market-economy-like system of supply and demand. Requests, built up out of several tasks, can be posed to the collective.

The goal is to gain a system able to perform simple tasks as well as very complex computations, while keeping the overall energy consumption low. This will be achieved by deploying highly specialized hardware accelerators and a reasonable resource management. First results show the viability of the methods in the distributed management of available resources.

Keywords: FPGA, Partial Reconfiguration, Distributed Computing, Self-Organization

Murr, D., Mühlbauer, F., Dressler, F., Bobda, C, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 373–386.

1. INTRODUCTION

The ongoing progress in chip technology provides us with steadily increasing computation power whereas the accompanying miniaturization shrinks the device sizes to unprecedented measures. Meanwhile electronic circuits are produced at rock-bottom prices. This trend has been around for so long that some are sure to have found a law in it. The new thing is that small, efficient, yet fairly powerful devices, are built that are able to communicate. This puts developers in the position of creating large and complex systems of interacting nodes that are small, powerful and energy efficient. With the computation power arises the ability of deploying highlevel-algorithms to implement some kind of intelligence into the networked units. This can be used to realize mechanisms of labor division and collective cooperation. Clearly this approach works if every specialized node behaves as specified by the designer at compile time. However, the system has no way to react to deviations due to failure-prone devices or changing operational and environmental conditions.

In order to enforce flexibility in such systems and allow single nodes to adapt their behavior to disturbances, we use reconfigurable units, in this case FPGAs. Those devices pose a decent trade off between computation power, energy consumption and flexibility. They can be configured and reconfigured to provide hardware acceleration for highly specialized services. Furthermore, they can be partially reconfigured while keeping the rest of the system working. Some are even capable of initiating their reconfiguration themselves keeping the part of their logical circuits that hosts the local operating system up and running while only a small partition that hosts specialized accelerators is being reconfigured. Despite those advantages a powerful and flexible management of available resources is required in order to optimize parameters like the overall power consumption in the system or a reasonable distribution of work among the nodes. In this work we present our developed approach for tackling this problem. A framework is built up that enables nodes to (re)distribute tasks in a marketplace-like manner that we called **LMGS** - *Local Marketplaces, Global Symbiosis* - which delivers the basis technology to elevate implementations of collective task completion to a new level. Here a node that recognizes the need for a certain task to be done formulates it as a query to itself and its neighbors. Every node that offers the execution of a task replies to a query with its cost for fulfilling the job. The inquirer is now able to choose whether it is more appropriate to maybe reconfigure and do the task by itself or to delegate.

An application field for this novel approach of a community of self-configuring nodes are sensor networks. Nowadays they are a prime example for a set of connected nodes that are bound to several restrictions: nodes are only powerful to a certain extend, energy is limited since simple sensors are battery driven and communication is costly. Still the data volumes to be processed in those networks are escalating not only when thinking of optical surveillance with recognition of biometrical features. We will show how to enhance the capabilities of such a network by deploying self-reconfigurable nodes as participants.

Next we point out some of the work that yields the foundation for our concept (part 2). Following we will go into the technological basis we've constructed in order to allow self-reconfigurable nodes (3), explain the ideas behind **LMGS** (4) and demonstrate how these two essentials can be combined to drive powerful sensor networks (5). Finally we summarize this article (6).

2. RELATED WORK

On the hardware side we want to concentrate on the application field of sensor networks. Here a variety of solutions has been constructed like the Mica2Dot Mote, the Tmote Sky or the iMote [2, 7, 5]. They particularly cause a low power consumption but in our case these systems are too weak concerning computation power. Their clock rate is as low as 10MHz compared to the racy Virtex-4 FPGAs that run at up to 500 MHz. Also their processors cannot be deployed as flexible as reconfigurable hardware since they are static and hard wired. Here FPGAs represent a much better compromise for our needs.

As a limitlessly flexible management system for resources we use Linux as operating system for the nodes. We built our own kernel and distribution on the basis of the PowerPC port of the Linux kernel [1] and MontaVista Linux [11]. For incorporating ICAP into the Linux system we rely on John Williams' device driver [12]. The partial reconfiguration is described in the main features in the Xilinx documents [14, 17].

The marketplaces idea took advantage of previous work like auction methods presented by Gerkey et. al [4], but we wanted to keep the algorithm much more simple to be able to deploy it on weak nodes with less computation power as well. The Open Agent Architecture [6] might be a viable approach for a very stable network without too much fluctuation and a central server that is very unlikely to stop working. We think the application field is much wider if we don't constrain our system to using only one specialized node. We moreover favor decentralized management and easy replacement of failing units. The aim is a system

small enough to run on the weakest units but also extremely expandable to allow sophisticated decision making when executed on powerful nodes. To estimate the energy consumption of transporting a certain amount of data over a connection energy aware routing protocols will be useful [9, 10]. The energy demand when operating a hardware module can be determined with [8, 13].

3. BASIC TECHNOLOGY AVAILABLE

The essential resources that must be available on each node in a distributed cooperative system are a minimal local computation power, a resource management, the ability to communicate and a system to distribute or gather jobs in the network. The own computation power allows nodes to locally solve tasks or to run more or less simple programs to decide whether jobs should be solved remotely. To be aware of the own capabilities and to easily deploy them there's necessity for a resource management that is flexible and generally applicable to run on most nodes in the network. From rudimentally equipped to all-in-one high-potential units. Communication allows cooperation to happen in order to spread parts of a bigger assignment that possibly cannot be solved by only one node to several others. Finally our framework **LMGS** provides a simple way to exploit the available resources, locally and remote, the most efficient way and to distribute large tasks among the contributors. It therefore realizes a variant of self-organization.

3.1 Computation Power in FPGAs

To develop and test our concepts we use a Virtex-II Pro FPGA by Xilinx sitting on the XUP development board as well as the Xilinx ML403 board equipped with a Virtex-4 FX12 FPGA [18, 16]. The FPGAs provide one resp. two embedded PowerPC hard-core RISC processors that are contained in the chip fabric. Besides the configurable logic cells, that allow custom hardware accelerators, the Virtex-II Pro and Virtex-4 both provide a certain amount of basic, hard-wired circuits (primitives) to extend the device's speed and effectiveness. These are for example multipliers, block RAM and especially a module named ICAP: Internal Configuration Access Port. As depicted in figure 1 this module is the key ability for a node to change its own reconfigurable logic. Additionally Xilinx FPGAs since the Virtex-II series are capable of partial reconfiguration. That means that single hardware

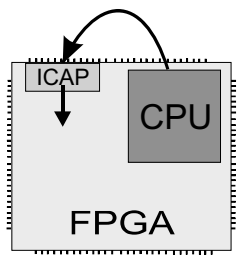


Figure 1. FPGA with ICAP-module

modules can be exchanged while vital parts of the cell like the memory controller or the network interface controller keep on operating uninterruptedly.

3.2 Local Resource Management

In order to cope with the increasing complexity that arises when managing a very large number of individual nodes we deploy a general purpose operating system. Linux serves our needs in several ways. It encapsulates the difficulties with accessing hardware and thus facilitates resource deployment through its abstract driver interfaces. For example the ICAP is incorporating into the Linux system with John Williams' device driver [12]. Especially the fully developed networking abilities are a convenient way to realize communication. The innumerable quantity of applications for any kind of need and furthermore adaption and development of software in high-level programming languages pose a big plus. The use of a standard Linux kernel and applications also speeds up development for another reason: the large community that exists and the vast database of solutions to standard problems.

3.3 Communication

As mentioned above our implementation enables standardized communication facilities. They provide us with a TCP/IP-stack to start with. Of course a certain application field might afford adapting the communication: A network that consists of battery driven nodes is more likely to yield better results when utilizing an energy aware routing protocol. The development boards we use provide RJ45- respectively USB-jacks, so besides the already implemented ethernet connection wireless LAN, bluetooth or ZigBee are feasible. For now remote nodes can be accessed and controlled via *telnet*. Data can be transferred via HTTP with *wget*.

3.4 Adaptive Processing

Using partially reconfigurable hardware devices we are in the position to equip our nodes with a virtually adaptive behavior. To keep development easy at the beginning our nodes provide only one region of fixed size that can be reconfigured separately from the rest of the system as shown in figure 4 on page 385. With this, we are able to realize a hardware-software co-design for optimized performance. Tasks that can efficiently be computed in hardware may be executed in a specialized hardware module. Other services that are not available as a hardware module or are more efficiently solved in software are then worked off

on the local CPU. We use the ICAP to allow the CPU to reconfigure the node itself to utilize free reconfigurable space or to displace unused hardware accelerators to save unnecessary electrical circuits and thus energy. The individual situational behavior of a node is determined by the controlling application which decides whether to replace a local hardware accelerator or to employ a neighboring node according to a set of parameters stored locally in simple files.

An example of such a parameter is the willingness of a node to reconfigure the local hardware to serve a remote nodes's request. Generally these parameters will be stored as relative quantities to system wide base values to keep them comparable. The parameter files can be written (adapted) to a changed environment, either by the control application itself or by another node. It is also possible to impose access control to prevent crucial data to be undeliberately or maliciously altered.

3.5 Self-Organization

The nodes, that are now able to change their own behavior to a certain extend, need to organize themselves in some way. We therefore developed a simple concept to deploy our new features: **LMGS** - *Local Marketplaces, Global Symbiosis*. The work is distributed according to principles of supply and demand within the network. As we touched on previously the controlling and resource management is left to the CPU with the Linux operating system and applications running on top of it. The *real* work to do in the network will also be done in software and, where possible, in specialized hardware. We present **LMGS** in more detail in the next section.

4. CONCEPT OF LMGS

The simple idea is that every node does exactly what it deems to be the *best* according to its stored parameters. The minimal software to actively 'take part in the game' consists of two elements. A **customer** which issues requests to the network to have a certain job done and a **purveyor** that answers requests for jobs with the costs it charged if it would be commissioned. The effort a node makes to create the answer can vary widely. According to its computation power, knowledge and storage capacity this can reach from a simple return of standard values to a complex measuring where the load, the utilization of the node's components or the probability for the effectiveness of anticipated reconfiguration might be taken into account. In sensor networks for example communication is the most expensive action so a distribution of

work should be chosen that minimizes overall traffic, maybe by executing a lot of tasks locally through reconfiguring the node.

4.1 Requests

Requests issued by the customer component of a node consist of a tuple as follows:

Request = {*Source, Target, Requester, Data_Volume, Task, Max_Hops*}

Where *Source* contains the data source for the task, *Target* the data sink and *Requester* the device which posted the inquiry. *Data_Volume* holds the quantum of data to be processed with *Task*. *Max_Hops* specifies the maximum number of times a request may be relayed by a node's neighbors.

Source, *Target* and *Requester* may be partly or completely identical. Given they are not, we included mechanisms to distribute jobs within a channel between data source and target. The values can be one-to-one identifiers of nodes or might as well contain wildcards to address groups of nodes so that, for example, modifications that have to be applied to a set of devices can be launched by a single command.

The *Data_Volume* will be taken into account when an offer is generated for the request. It usually influences the decision whether it is more appropriate to solve a task in software or in hardware and if the data may be processed remotely or if the communication costs for that would be too high.

In order to be able to specify a *Task* or a whole group of tasks, we suggest an hierarchical nomenclature which comprises every single service that can be rendered in the network under one root node.

As *Task* in the request structure is not limited to one atomic job it may be a list of tasks. These lists may contain jobs that are to be processed sequentially or in parallel reaching from a single data-source to a single target, to complex data flows with multiple sources and targets.

Finally the restriction to *Max_Hops* ensures that inquiries are not simply flooded through the net but stay local working off jobs at a kind of in situ marketplace when sensible. This of course only if the local neighbors provide appropriate solutions to tasks at a reasonable price. The composition of this 'price' will be presented in section 4.2. The idea behind that is obvious: since we are able to reconfigure nodes to serve virtually any need, even small localized groups are highly adaptable and will be able to cope with most challenges with optimal efficiency. Thus data will in general be kept in a spatially narrow cloud and communication is minimized. Later in this section and in part 4.2 we will explain how this is accordable with the claim of super-regional cooperation.

To publish and find services to fill in a valid request basically three mechanisms can be deployed. First one central directory server keeps track which node offers which services and has to be prompted for every job to work off. If it fails the whole network is paralyzed. New services have to be entered, causing additional traffic. The inherent single point of failure may be compensated by mirrors but also causes additional costs for synchronization. The nodes surrounding a server are exposed to heavy communication traffic. Bottlenecks and fast resource exhaustion are preassigned.

Second, searches for certain services are flooded through the whole network in a very flexible, but rather inefficient way. Nodes respond to a request if they find themselves in the position to satisfy it. The disadvantage is that information is being spread redundantly to all of the nodes even if the best candidate is just one relay apart from the issuer. This depletes the resources of the complete network needlessly. The advantage is a superior flexibility and reliability. Nodes can knock off at idle times or be switched on in phases of high load to aid the network.

In this work, we developed a third approach, a hybrid solution between the two previously mentioned ones. Here data is flooded only within the closest neighborhood. Only if the answers from them are insufficient, say because none of the next nodes wants to execute the requested task, the job is advertised again, this time with a higher number of maximum relays. Additionally nodes keep a more or less extensive list of other, remote hosts, that satisfy a certain request. In this manner not only local offers will be taken into account but also more distant ones which possibly better suit the current situation. The maintenance of this list is closely related to the structure of offers replied to a request which will be covered in section 4.2.

Generally, requests will be posed to the direct neighbors and to the issuer itself. A neighbor that finds the *Max_Hops* greater than one reduces that counter and sends the request to all it's neighbors. Especially the answer from the node itself is interesting in this regard: with the possibility to reconfigure, scenarios can be managed where communication cost is very high and nodes have to cut back on transporting lots of data through the net.

4.2 Offers

The response to a concrete query contains two elements: the cost-vector that the replying node is estimating for supplying the service and

the local list of known providers that are also capable of satisfying this particular request: $Offer = \{CostVector,\ List\_of\_known\_providers\}$

Costs mean figures given as multipliers of a base cost. For example the transmission of one byte of data over a wireless bluetooth connection will be a lot more expensive than over a wired ethernet link.

We identified three dimensions of costly actions: time consuming, energy consuming and space consuming. Thus a node's purveyor calculates the cost vector for a task A according to

$$C_A = \begin{pmatrix} Z_K & E_K & P_K \end{pmatrix}^T \cdot W$$

where Z_K denotes the cost for the local effort concerning time, E_K for energy and P_K or required space. W contains the willingness of the node to spend part of the specific resource to locally execute task A as a diagonal matrix. A battery driven node might for example want to lower its willingness to accept very energy consuming jobs as it runs low on battery power. The final cost-vector C_A is passed to the issuer as part of the offer.

The list of additional service providers is being built up through logging of messages indicating the commission of a node for a particular task or through deliberate writing. On the one hand devices that relay an acceptation message (basically a request with a specially formulated task) to a node store the target node and task together with an expiration time. On the other hand nodes can advertise their capabilities by giving out a request to every other participant of the net to amend its local list of providers. If it intends to stay in the community for a longer period the expiration time may be set accordingly, attracting all sorts of requesters, locally and remote.

4.3 Negotiation Example

The flow of a negotiation bases on sending requests, retrieving responses, determining the most appropriate solution and commissioning a purveyor (figure 2 on the next page). When evaluating the replies, the contained lists of alternative, maybe remote, providers may be taken into account and selected ones may be prompted for a bid. This enables the system to incorporate both: decentral, distributed computing as well as central services like the storage of gathered and processed data.

When enough answers came back in or after an amount of time the *customer* determines the optimal partner to commission. The weighted cost including communication is therefore calculated according to

$$A : CC_A = \vec{C_A} \cdot \vec{G_A} = \textstyle\sum_i C_{A_i} \cdot G_{A_i}$$

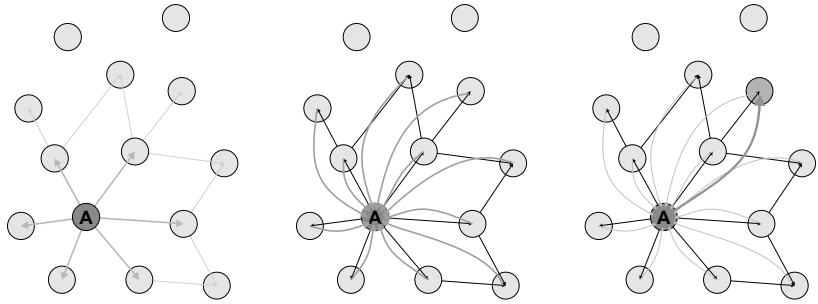


Figure 2. **LMGS**: issue a request (left picture), retrieve answers (middle) and commission job (right)

Manipulating the components of G_A allows the node to emphasize a rather fast, energy-efficient or space preserving execution of a task. Depending on the computation-power and -willingness the node might accept the earliest offer or may run a multi-goal optimization on the plenty of data retrieved to find the best trade off.

5. ADAPTION TO SENSOR NETWORKS

Sensor networks used to consist out of many very small sensor nodes to collect data and at least one data sink that gathers all the information and offers an interface to other nets. In scenarios with more complex data aggregations beacons are introduced which provide more computation power, collect data and transmit preprocessed information to the destination.

Our targeted application is a sensor network that is able to track an object optically over a large area where several cameras have to cooperate to keep the target within sight. A sensor node is now equipped with a video camera and has a certain computation power to extract features for object recognition from the taken pictures. Information about what the cameras detect will be send to the user's terminal. If the nodes don't provide enough computation power to cope with the load the tasks involve additional processing nodes should be added easily. In the ideal case the system now distributes the work differently, deploying the capabilities of the new nodes.

For this we built up a sample application for the XUP development board that captures the video signal from a camera, digitizes it and applies a filter to the stream to provide it for further processing as depicted in figure 3 on the facing page. To start with we have implemented a mean-value filter to suppress noise, a sharpen filter and a set of Sobel filters for edge detection in x- and y-direction.

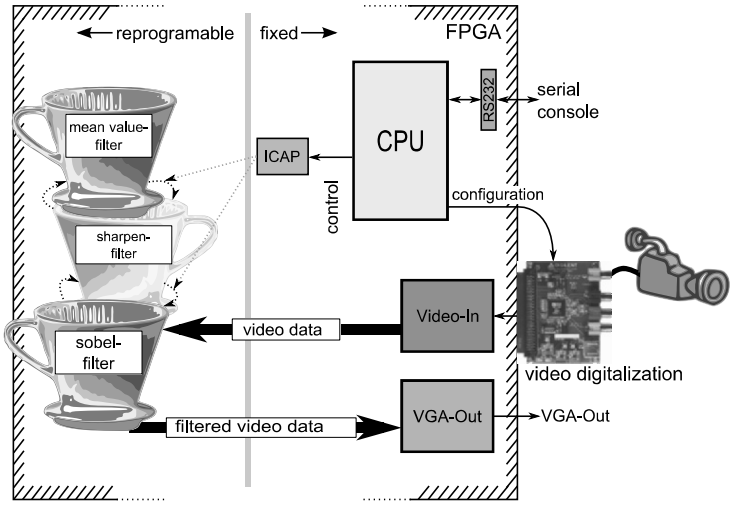


Figure 3. Scheme of our sample system: the video stream is digitized with a Digilent VDEC1 expansion card [3].

The whole task is solved in hardware on the FPGA with the filter being partially reconfigurable at runtime. A program running on the CPU, in our case a PowerPC 405, configures and controlls the ICAP and the video digitizer module, loads a partial bitstream and initiates the reconfiguration of the filter. In our implementation the program controll is accessible over the serial port to present status and debugging information.

Figure 4 on page 385 shows an overview (left hand side) and the layout (right) of the physical implementation of our sample system. There the small boxed area is reconfigurable, whereas the bigger region encloses fixed logic like the ethernet controller or the SD-RAM controller. Both are connected via fixed interfaces, so called Bus Macros. The lighter lines on the right picture are wire connections between logic blocks.

We evolved our designs to be partially reconfigurable using Xilinx's Early Access Partial Reconfiguration design flow [14] and made use of tools like PlanAhead, FPGA Editor and EDK [15].

Surely much of the video processing might be solved in hard core graphic processing units as well. Sure the programming of these is easier and more standardized than partial reconfigurable module implementation for FPGAs but flexibility and energy efficiency are the unbeatable advantages of our approach. FPGA nodes not only support hardware acceleration for graphic processing but also for virtually any task that is realizable in hardware. Hardware modules can be implemented using

a hardware description language like VHDL and thus are relatively easy to develop. Additionally, unused hard core units still waste space and cause stray current whereas unused FPGA modules can be physically disconnected from the current-carrying system, replaced by an other needed component or erased completely. This reduces the overall power consumption for a complete system with certain abilities that are on the one hand constructed out of many specialized hard core hardware components, on the other hand using one extremely flexible FPGA.

Using Linux several power saving modes like suspend to memory or to disk are feasible. This helps reducing the power consumption of the CPU when it's processing capabilities are not demanded. The node can even be set to sleep mode with only few circuits one to listen to a wake-up message.

LMGS is designed to be deployable on devices from both ends of the spectrum: very powerful or small, energy preserving nodes. In a sensor network the minimal equipment with **customer** and **purveyor** can even be further tapered: a sensor node generally doesn't have to issue requests thus running a **purveyor** module on it is enough to be able to obtain its measurements. The node simply answers to corresponding requests and will only be charged with a minimal amount of computations.

Since our system is flexible and expandable the loss of camera or computation nodes (beacons) can be compensated to a certain degree in terms of surveilled area as well as computation power for picture analysis. This is achieved by distributing the work the failing node contributed to other units.

The data needed to configure a possibly missing hardware module into a neighboring node can be obtained from a central storage or, better when thinking of reliability, from a node that still has it in his cache. Using such a cache for hardware module definition files also serves another need: data that is used more often can be retrieved from a lot of nodes maybe even one in the vicinity of the own location thus the time to get and deploy such a module may be reduced. Also new camera nodes or beacons can be added as the task of the network and therefore the demand for computation power changes.

6. CONCLUSION

This excerpt of our work presents our approach on how to deploy (partially) reconfigurable hardware in a network of semi-intelligent nodes to optimize overall resource demand and increase stability. It yields to a more optimal resource usage because we keep tasks local were appropriate and spare expensive communication. Stability is gained because

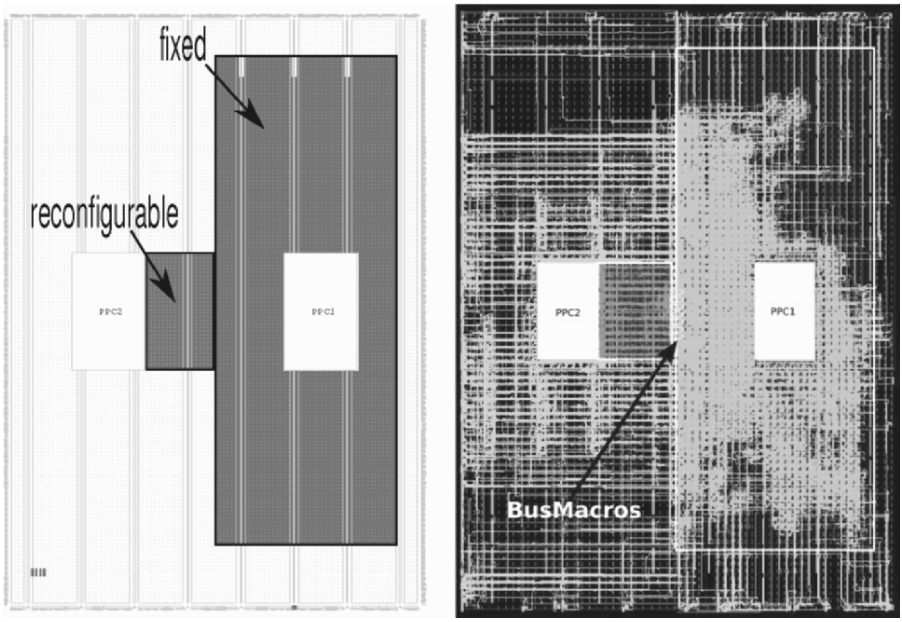


Figure 4. Layout of our sample system: the small emphasized area is reconfigurable.

services that have been offered by nodes that just dropped out will be distributed among the neighboring nodes. They will be prompted to reconfigure themselves and are then able to compensate the failing node as a community. For a sensitive distribution of work in the net we introduced **LMGS** which relies on the concepts of supply and demand. We created a working sample system for the Xilinx Virtex-II Pro XUP development board that allows us to initiate partial self-reconfiguration, send and receive files, control distant nodes, being controlled remotely and to deploy standard applications. The local resource management is assigned to a Linux operating system, thus governed in software. Also high-level applications like load balancing of the computations and the network congestion or determining the node which suits the needs of our task the best are designed in software. The real work to do in the network, besides the administrative clutter, can now be accomplished using a hardware-software co-design. The filtering of a video stream or the detection of keywords in an audio file can now be split up in parallel and sequential parts and we are able to have the parallelizable parts run in one or more hardware modules. Still some tasks will be solved in software but the increase in performance is significant.

REFERENCES

[1] Linux on ppc. http://penguinppc.org/.

[2] Crossbow Technology. The mica2dot mote. http://xbow.com/Products/-Product_pdf_files/Wireless_pdf/MICA2DOT_Datasheet.pdf.

[3] Digilent Inc. The vdec1 video decoder board.

[4] Brian P. Gerkey and Maja J Matarić. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation*, 18(5):758–768, October 2002.

[5] Intel. imote isn100-ba data sheet.

[6] D. Martin, A. Cheyer, and D. Moran. The Open Agent Architecture: a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2):91–128, 1999.

[7] moteiv. Tmote sky homepage.

[8] Jingzhao Ou and Viktor K. Prasanna. Rapid energy estimation of computations on fpga based soft processors. In *IEEE International SoC Conference (SOCC)*, 2004.

[9] R. Shah and J. Rabaey. Energy aware routing for low energy ad hoc sensor networks, 2002.

[10] Suresh Singh, Mike Woo, and C. S. Raghavendra. Power-aware routing in mobile ad hoc networks. In *Mobile Computing and Networking*, pages 181–190, 1998.

[11] MontaVista Software. Montavista linux professional 3.1. http://www.mvista.com/.

[12] John W. Williams and Neil Bergmann. Embedded linux as a platform for dynamically self-reconfiguring systems-on-chip. In Toomas P. Plaks, editor, *ERSA04: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 163–169, Las Vegas, Nevada, USA, June 2004. CSREA Press.

[13] Xilinx. Xpower. http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm.

[14] Xilinx Inc. Early access partial reconfiguration user guide. Xilinx User Guide UG208, Version 1.1, March 6, 2006.

[15] Xilinx Inc. Homepage. http://www.xilinx.com/.

[16] Xilinx Inc. Ml403 evaluation platform. http://www.xilinx.com/products/boards/ml403/docs.htm.

[17] Xilinx Inc. Two flows for partial reconfiguration: Module based or difference based. Xilinx Application Note XAPP290, Version 1.1, 2003.

[18] Xilinx Inc. Virtex-ii pro xup development board. http://www.xilinx.com/univ/xupv2p.html.

DYNAMIC SOFTWARE UPDATE OF RESOURCE-CONSTRAINED DISTRIBUTED EMBEDDED SYSTEMS

Meik Felser, Rüdiger Kapitza,
Jürgen Kleinöder, Wolfgang Schröder-Preikschat
Friedrich-Alexander University of Erlangen–Nuremberg
Dept. of Computer Sciences 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
E-Mail: {felser, kapitza, jk, wosch}@cs.fau.de

Abstract: Changing demands, software evolution, and bug fixes require the possibility to update applications as well as system software of embedded devices. Systems that perform updates of resource-constrained nodes are available, but most approaches require a complete restart of the node after installing or updating software. Restarting the node results in the loss of important system state, such as routing information or sensor calibration values. Rebuilding this information requires time and energy.

In this paper we present an online state-preserving update system for resource-constrained nodes. A remote incremental linking approach is used to generate node-specific and execution-state dependent code. Compiler-generated symbol, relocation, and debugging information is used to determine whether a dynamic update of the running system is possible and how it can be achieved.

Keywords: dynamic update, managed nodes, ELF, relocations

1. INTRODUCTION

The number of microcontrollers used in today's embedded systems is increasing. In more complex systems, a network of several different microcontrollers is used, thereby forming a heterogeneous system of larger and smaller nodes. Especially, we have to face distributed embedded systems with a mixture of very resource-constrained devices on the one hand and more powerful devices on the other hand.

An example of such a network is a sensor network [1], a large number of small, non-expensive, and very resource-constrained nodes equipped with sensors to collect environmental data and able to communicate. Usually the sensor

Felser, M., Kapitza, R., Kleinöder, J., Shröder-Preikschat, W, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 387–400.

nodes are supported and managed by a larger, more powerful node, the base station.

Once initially set up, selective software updates will become necessary due to bug fixes, improved functions, extensions, and parameter changes. For example, if the calibration process did not deliver a set of parameters that yields the desired results, software modifications become necessary. Replacing parts, such as the sampling or pre-processing algorithm, or adding additional software to use another sensor will be the consequence. Furthermore, with an estimated lifetime of several years, a future-proof system must be able to replace any part of the system such as the scheduling system or the communication protocol.

When updating the software of a node it is desirable to do this with as little impact as possible. Considering the limited resources of sensor nodes, the update process itself must not use a lot of resources. Furthermore, different updates affect different parts of the system and updates of the application should not affect system services. It is not acceptable that the addition of a sensor driver results in the restart of the complete node. When restarting the node, state information of applications and the operating system library is lost. Multihop communication protocols, for example, store routing information to represent the network topology. After a restart such information is lost and has to be rebuilt. The routing of the network has to be redetermined, which does not only affect the local node but the communication of other nodes, too.

Our goal is to update sensor nodes dynamically thereby preserving application and system state. During the update process the nodes are stopped but no restart is needed after the update is complete. The advantage is a considerable shorter off-line time and a better performance resulting from a very quick recovery after the update because the system state does not have to be rebuilt unnecessarily.

We describe an update infrastructure that allows dynamic reprogramming of nodes based on the binary code of the application. Most parts of our update system are located at a more powerful node, the base station for example. We use compiler-generated information, such as symbol tables, relocation tables, and debugging information, to identify situations in which a safe update is possible. In rare cases, the execution state might prevent a dynamic update. Such situations are signalled to the administrator so that additional actions can be taken, such as a redesign of the application to increase the updatability.

The following section gives an overview about related approaches for updating nodes in sensor networks. Section 3 outlines the architecture of our system. After that the object file analysis is described followed by section 5 presenting the procedure for updating or replacing code. Section 6 concludes the paper and gives a short overview of the current prototype.

2. RELATED WORK

There are several approaches for dynamically updating software. In the following we will discuss some existing approaches for updating sensor nodes.

TinyOS [7] provides XNP [2, 10] to update a sensor node. This approach only supports very coarse-grained updates because the complete memory image is replaced. It has a huge impact if only some parameters are to be adjusted. Deluge [8] provides similar possibilities and uses Trickle [13] as multihop dissemination protocol to distribute the update to multiple sensor nodes. Both approaches need to restart the sensor node, because a new memory image is installed.

FlexCup [15] is the update system of TinyCubus [16], a framework to configure nodes in a role-based way. Components are transmitted as relocatable binary images including symbol tables and relocation information. A linker component on each node integrates the new components into one image. The system is restarted after the update to eliminate dangling pointers. Our approach does not need the linker component on each node because that part of the process is performed remotely.

Koshy and Pandey [11] present an approach similar to ours. They modify the development toolchain to create an incremental linker that is used to prepare the updated code on the base station. Thus, they can control where modified functions are to be placed on the node and reduce modifications resulting from moved code. They do not consider the current state of the system and perform a restart after the updated code is written into flash memory.

There are projects that support updating of a running system. To find all references to the code they often use an indirection layer to access the updatable code. One example is SOS [5], an operating system for sensor nodes that is built of modules. Modules can be updated, removed, and replaced at run time. The modules are transmitted and installed in a relocatable binary image format. Module code is position-independent by using only relative jumps, thus limiting the maximum size of a module to 4 Kbytes, the maximum distance of relative jumps on the target platform, the AVR ATmega128. Furthermore, references to functions or data outside of the module are implemented via an indirection table or are not allowed, respectively. Contiki [3] works similar and also provides a framework for dynamically loading libraries. The libraries contain relocation information that enables the installation of the code. Access to the libraries is provided indirectly via stubs. Our approach does not rely on an indirection table; we identify and modify the references directly. With this approach, we can even update code of the operating system library or the kernel.

3. ARCHITECTURE

To achieve a minimal update infrastructure on the resource-constrained node, most parts of the update preparations are executed by a remote node. That node is called the *manager* and is usually more powerful and equipped with several megabytes of memory. The manager node is responsible for updating and managing the software installed on the resource-limited nodes (Fig. 1).

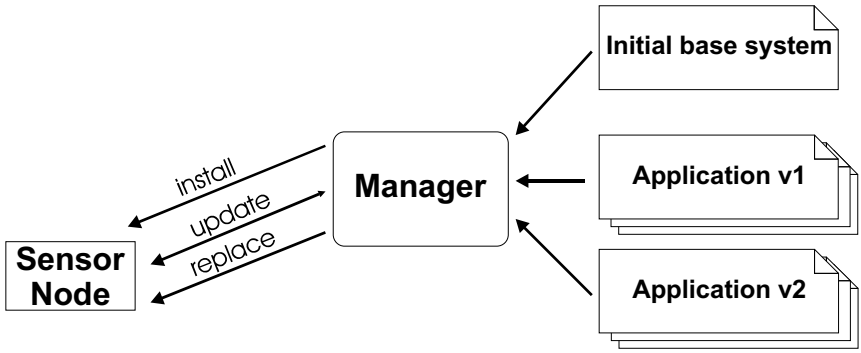


Figure 1. Duties and responsibilities of the *manager*.

As we will discuss in section 5 some situations preserve the automatic update of the running system. The information automatically extracted is not sufficient to decide whether the update of the running system can be performed. In these cases the administrator, who runs and supervises the system, is informed. He can the take the decision or use the information to redesign the application in a way that the manager can decide the situation automatically the next time.

Figure 2 shows a schematic overview of the manager node. The next paragraphs give an overview about the most important elements.

3.1 Repository Manager

The *repository manager* is responsible for managing and analysing the code. The code is provided as binary object files. Software that is already installed and currently running on a sensor node is called the initial image of a node. The repository manager determines which functions, more precisely which symbols, are available in the compiled and linked ELF [22] object file of the initial image. Software that is to be updated or installed on a node has to be provided as *relocatable* ELF object files.

The *ELFExtractor* loads these binary files and analyzes the contents of each input file to identify individual functions and data objects. This results in a fine-grained modularization of the application. After that the repository manager extracts the dependencies between the identified objects from the reloca-

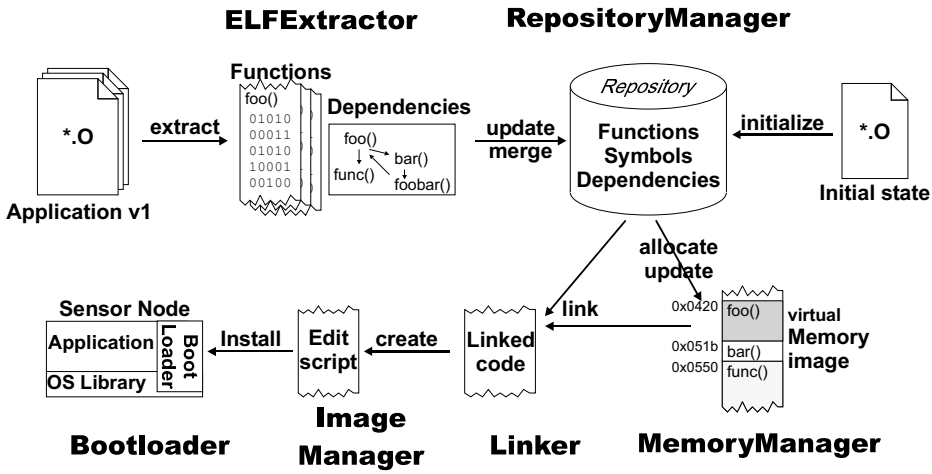


Figure 2. Architecture of the manager node.

tion information and builds the dependency graph to determine which data is needed by a function and which other functions are called. More details about the modularization process are given in section 4.

When the administrator changes a function the system identifies which functions and data in the dependency graph are affected and where these parts are installed in the network. From this analysis a set of differences for each node can be calculated. This set of changed functions and their dependencies present the basis of semi-automatic policies that are used to determine whether the update operation can be performed. In case of conflicts the administrator gets involved to control the update operation. This is addressed in detail in section 5.

3.2 Image Manager and Memory Manager

The *image manager* creates and manages a memory image model that represents the usage of the memory in the device. At first this memory model is initialized with the initial state of the node. The model is then used for keeping track of the currently installed software layout. Based on this information the *memory manager*, a part of the cross linker, determines a location where to put modified code on the node. For example, an updated function should be located at the same position as the original function to minimize the number of references that need to be updated. The virtual image includes a list of application modules that are currently installed on the node including their exact position.

The image manager also records changes the linker performs to update the application. Based on this information it calculates a change set after the new functions were linked into the virtual image. This set contains the differences for each updated function and the code of all new functions. Deleted functions are not in the change set. The system ensures that these functions are not used anymore and the memory manager reuses their space to store new functions. The changes can then be transmitted to the node in form of edit commands as they are generated by common diff algorithms, such as UNIX diff [9] or Xdelta [14].

3.3 Bootloader

A small bootloader resides on each node and is able to receive and process commands from the manager. Its main task is installing new code but it also provides the manager with essential information about the current state of the node. As described in section 5 the current state influences whether an update of the running system is possible or not.

When a sensor node receives an edit command, the application transfers control to the bootloader which either extracts the requested data or modifies the SRAM and flash memory of the sensor node.

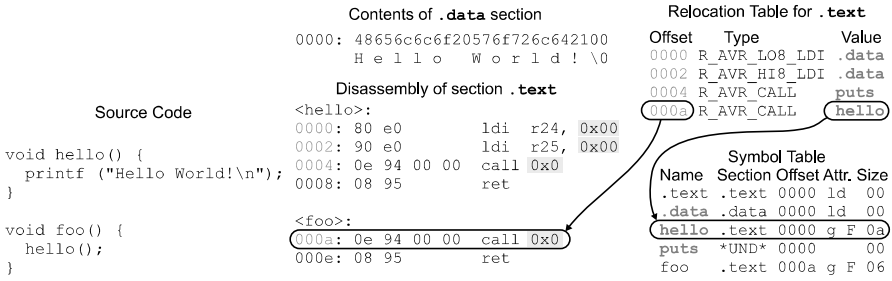
The actual transfer of code blocks can be done via several protocols. Either direct single-hop communication or more complex multihop protocols like Trickle [13] or MOAP [20] are an option.

4. MODULARISATION AND DEPENDENCY ANALYSIS

Functions and data objects are identified by analysing the relocatable ELF object files. The analysis is solely based on the information provided in this ELF files. The developer does not need to use special constructs or annotations to support the analysis.

ELF files are organized in sections. Some sections contain symbol tables and relocation tables, other sections contain the relocatable binary data. A simple example is shown in figure 3. To determine the dependencies, we use the symbol and relocation information an ELF object file provides. For each relocation the position inside the associated binary section is given at which the final address of a symbol should be inserted. The final address of a symbol is resolved by the linker as soon as the location of the associated section inside the target's address space is determined [12].

To extract a function we look at the symbols associated with the `.text` section. These symbols give us the name, the start offset, and the size of the code. Thus we know where each function starts and how much space it allocates. To determine the dependencies of a function we take advantage of the



(a) source code (b) ELF file contents

Figure 3. Example of the data contained in an ELF file and its relationship

relocation table. Each relocation that affects the code of the function represents a dependency.

We instruct the compiler to place each function in a separate section in the object file. This way we ensure that each function is visible and each dependency is represented by a relocation embedded in the ELF file. Otherwise calls of local (`static`) functions might not be visible in the relocation table. In the example in figure 3 both functions are in the same section. Thus the compiler knows the distance between the two functions and the call from `foo` to `hello` could be done via a relative call without a relocation entry. If `hello` is additionally static, the compiler might even suppress the symbol.

We do not necessarily have to apply the same approach for the data objects of a program, because they are never placed in the same section as the functions. Thus, the symbols, even of local data, are available because the compiler needs them to insert references to data. Nevertheless, we instruct the compiler to generate separate sections because this makes it easier to identify each single data object. Otherwise the compiler creates one symbol for several local variables and just uses different addends and a platform-specific analysis would be necessary to separate such data objects.

5. DYNAMIC CODE UPDATE

Replacing or updating code in a running system requires knowledge about all references to the old code and their substitution with references to the new code. We also have to adjust the target addresses of jumps if a function is moved to another address. We further have to take special care if the function is modified while it is used by a thread of control. An overview of problems and possible solutions when updating software dynamically is given in [6]. Here we address three problem classes:

- references to a function as it occurs in function calls
- active functions that are currently executed by the target
- functions on the call stack (the function called a subfunction that is currently active)
- update of global variables

In the following subsections we examine and discuss these problems.

5.1 References to Functions

The first question we have to address is whether the updated function starts at the same address as the old one. If the new code has the same size or is smaller than the original function it fits into the old space and can start at the same address. This is the preferable situation because we do not need to change any references. To allow even larger updates fit into the space of a function the memory manager may allocate extra space [18] when initially placing a function. In resource-constrained environments this approach has to be applied carefully, because the additional space is wasted as long as it is not used by a replaced function.

If the modified function is larger and does not fit into the space of the old function it has to be installed at a different location. Then we have to identify all references to this function and update them.

References to the start of a function are used in conjunction with calls. Ordinary function calls can be detected by the use of symbol and relocation information. The relocation table for a function that calls another function contains an entry specifying the location where the address of the target function must be inserted (fig. 4). We use this information to find and update the reference to the function.

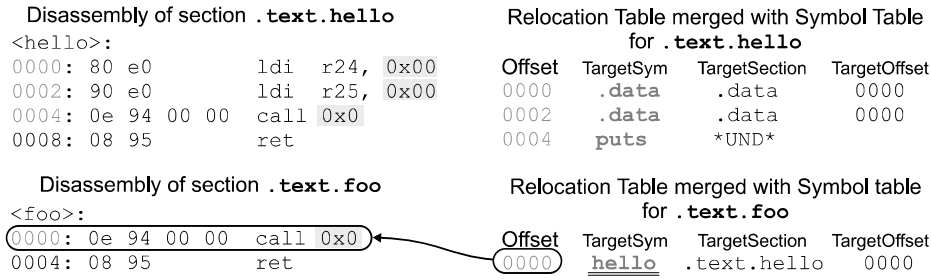


Figure 4. The relocation information shows that `foo` needs `hello`. If we update `hello` and thereby move it to an new position we need to modify `foo` as well.

Before patching the location with the new address, we have to determine how the address is used. It may either be used in a call operation or it may be stored in a variable to do an indirect call at a later time. We can easily patch the call operations with the new address. In all other cases it can not be exactly determined when and by which function the address will be used. In the worst case the address of the function is stored in a global variable for use by other functions. A combined code and data flow analysis could be able to detect this case but we do not think that the cost for this highly architecture-dependant operation is justified.

Our approach is to identify whether there is enough information to correctly find and patch all references. As a consequence we cannot patch calls via function pointers and, thus, forbid the use of function pointers, at least for functions that should be updated. Nevertheless, function pointers are allowed in well-known code, such as interrupt vector tables and the operating system scheduler. A system-specific layer is used to identify code and data references in this code.

This is not a strong limitation as function pointers are very rarely used in embedded applications[1]. Furthermore, function pointers are an additional source of errors especially for inexperienced programmers. For the same reason other programming paradigms for embedded systems, such as the Misra C rules [17], forbid the use of function pointers at all.

If our system detects the address of the updated function in any other than a call operation it indicates an insecure situation to the administrator who has to decide whether the function should be updated nevertheless.

5.2 Active Function

Before actually updating a function we have to make sure that it is not currently in use. The consequences of such an update are unpredictable. A function is in use if it is interrupted by the update process, as shown in figure 5, or if a thread was executing the function before it was suspended. In some event-driven systems, such as TinyOS [7], this situation does not occur as they do not offer a thread concept. Just event handlers can interrupt the normal control flow. Thus the update process gets scheduled as a task when no other task is active. Other systems, such as Contiki [3] or Nut/OS [4] offer a thread concept.

To identify active functions, the manager asks the bootloader at which address the system would resume operation when exiting the loader. If the system supports multiple threads the bootloader returns the current position of all

[1] An analysis of typical TinyOS applications showed that usually no function pointers are used. Nevertheless pointers to functions are used by the TinyOS scheduler and implicitly in the interrupt vector table.

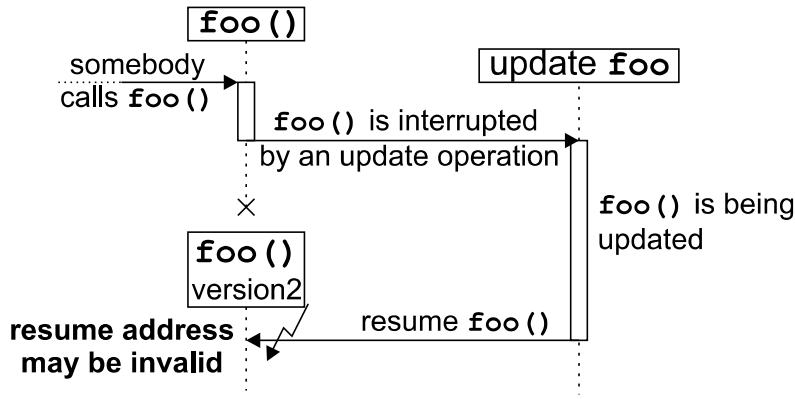


Figure 5. A function is interrupted and updated while it is active. Resuming the execution at the interrupted address may result in a crash or unpredictable behaviour.

threads. The update manager checks whether the update would affect one of these addresses and indicates an update problem.

A simple but promising approach to overcome this situation is to resume the system and retry the update a few moments later. If we are updating a sensor reading function it is likely that the function is not in use the next time we try an update. To avoid the periodic check, we can improve this approach by modifying the return address in a way that the bootloader is called as soon as the function returns. We then resume normal operation and wait until the function returns into the bootloader. If the function is still in use after a timeout or several retries, we inform the administrator about the failed update. Then he can authorize a reset after the update.

The success of this approach depends on the modularity of the application. Sensor net applications developed with some sort of framework, like TinyOS, are often designed very modular. Furthermore we inform the administrator about the reasons of a failed update. He can use this information to improve the application's modularity with regard to the next update.

5.3 Functions on the Call Stack

Beside functions that are directly in use by a thread, functions may also be on the call stack. That is, the function that is to be replaced called another function and this function is currently active. When the called function returns it uses the return address on the stack to resume operation in the caller. If we replace the code of the caller this return address might become invalid. Figure 6 shows an example. `foo` calls `hello`, this may be a call from a sensor calibration function to the function that actually gets the sensor readings. During the execution of `hello` the function `foo` is updated. `hello` is not influenced

directly but the code of `foo` is modified and the return address that is used to resume `foo` after `hello` returns may be incorrect or invalid.

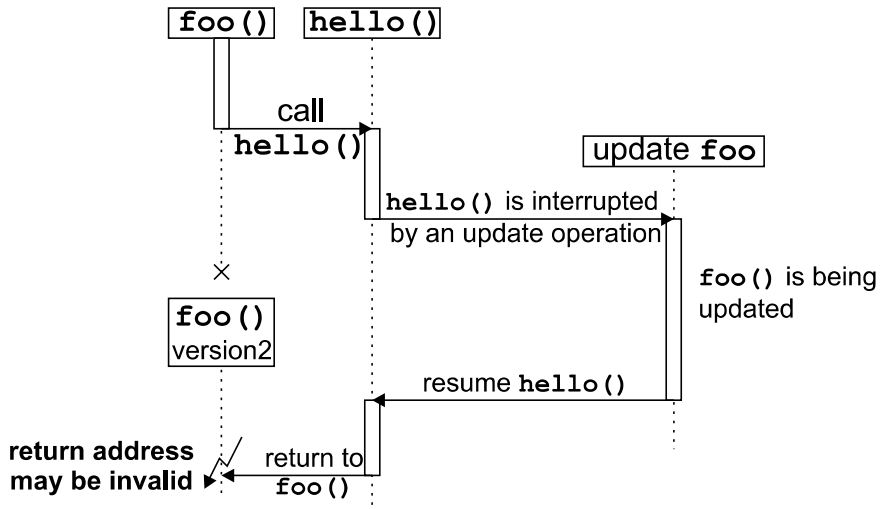


Figure 6. A function on the call stack is updated. When the flow of control returns to the updated function the resume address may be invalid.

We do not want to forbid the update of such functions in general, because minor changes, such as bug fixes, may alter the addresses but not the functionality of the code. Thus, updates are allowed if they do not alter the structure of the function. The new code must have the same calls to the same subfunctions as the old code and the same local variables. In our example above, we can fix a bug in the calibration algorithm `foo` as long as we still call the reading function `hello` and do not insert or delete local variables. We then assume that the return address A of the n-th call to a specific function from the old code can be replaced with the equivalent return address A' in the new code.

To accomplish this we have to know the stack layout and position. With this information we can walk through the stack and detect each function on the stack. The return addresses can then be adapted. The information how to obtain the stacks is encoded in the bootloader. The manager transfers a mapping from old to new return addresses if the loader requests it.

The debug information can give us information about the local variables of a function. Thus we can perform an offline check whether new local variables were added or removed.

If the structure of the function is modified we can retry the update several times. The function is possibly not in use anymore at a future retry. But the deeper the function is on the call stack the smaller is the probability that the function is not in use. If the retries are not successful, we inform the

administrator about the failed update. Again, he can authorize a reset and use this information for the development of future versions of the application.

5.4 Update of Variables

Update of local variables is not supported as mentioned in the previous sections. Updates of global variables are allowed under certain conditions. If a function references a new global variable, this variable is simply allocated on the node. In the opposite case, that a variable is no longer used by an updated function we can not reliably determine whether the variable is used by some other code. Therefore we ask the administrator if the variable can be freed if we do not detect any references anymore. If a variable with the same name but different type is introduced we have several possibilities. If the old variable is still used by some code, we generate a warning because this situation most likely leads to inconsistencies and the administrator has to decide whether to continue with the update or not. If the new variable replaces the old the current state should be transformed. Without further information from the administrator this is not possible. An automatic transformation is subject to the same restrictions as, for example, the object (de-)serialization in Java [21].

6. CONCLUSIONS AND CURRENT STATUS

The presented work builds a cornerstone of our long-term objective to provide support for robust and efficient software management in heterogeneous sensor networks [19]. It enables the management of software installed on resource-constrained nodes by providing the ability to update and replace code in the running system. This is achieved by incrementally link new code to the existing application and to identify unused code. Necessary configuration information is stored at a larger node that prepares the update. This manager node also checks whether an update is possible in the running system at all. In critical situations, the administrator gets involved who can control and decide in which way the update has to be performed.

The current prototype of the manager software is implemented in Java. It can load and analyze relocatable ELF object files for x86, Hitachi H8 and AVR CPUs. Up to now, the prototype communicates and updates single nodes but it will be extended to handle groups of equal or similar nodes.

REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, Mar. 2002.

[2] Crossbow Technology, Inc. *Mote In-Network Programming User Reference*, version 20030315 edition, 2003.

[3] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th IEEE Int. Conf. on Local Computer Networks (LCN'04)*, pages 455–462, Nov. 2004.

[4] egnite Software GmbH. *Ethernut Software Manuals*, Nov. 2005.

[5] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *3rd Int. Conf. on Mobile Systems, Applications, and Services (Mobisys '05)*, pages 163–176, June 2005.

[6] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, Aug. 2001.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Nov. 2000.

[8] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *2nd Int. Conf. on Embedded Networked Sensor Systems (SenSys'04)*, pages 81–94, Nov. 2004.

[9] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report 41, Bell Telephone Laboratories, 1976.

[10] J. Jeong, S. Kim, and A. Broad. Network reprogramming. Technical report, University of California at Berkeley, Aug. 2003.

[11] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *2nd Europ. W'shop on Wireless Sensor Networks (EWSN 2005)*, pages 354–365, 2005.

[12] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, USA, Oct. 1999.

[13] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *1st Symp. on Networked System Design and Implementation (NSDI '04)*, pages 15–28, Mar. 2004.

[14] J. P. MacDonald. File system support for delta compression. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2000.

[15] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *3rd Europ. W'shop on Wireless Sensor Networks (EWSN 2006)*, volume 3868 of *LNCS*, pages 212–227. Springer, Feb. 2006.

[16] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A Flexible and Adaptive Framework for Sensor Networks. In *2nd Europ. W'shop on Wireless Sensor Networks (EWSN 2005)*, pages 278–289, Jan. 2005.

[17] MISRA. *MISRA-C: 2004 - Guidelines for the use of the C language in critical systems*, Oct. 14, 2004.

[18] R. W. Quong and M. A. Linton. Linking programs incrementally. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):1–20, Jan. 1991.

[19] W. Schröder-Preikschat, R. Kapitza, J. Kleinöder, M. Felser, K. Karmeier, T. H. Labella, and F. Dressler. Robust and efficient software management in sensor networks. In *2nd IEEE/ACM International Workshop on Software for Sensor Networks (SensorWare 2007)*, Jan. 2007.

[20] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS–TR–30, University of California, Los Angeles, Center for Embedded Networked Computing, Nov. 2003.

[21] Sun Microsystems. *Java Object Serialization Specification*, 2001.

[22] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, version 1.2 edition, May 1995.

CONFIGURABLE MEDIUM ACCESS CONTROL FOR WIRELESS SENSOR NETWORKS

Lucas F. Wanner, Augusto B. de Oliveira, and Antônio A. Fröhlich

Laboratory for Software and Hardware Integration
Federal University of Santa Catarina
PO Box 476 – 88049-900 – Florianópolis, SC, Brazil
{*lucas, augusto, guto*}*@lisha.ufsc.br*
http://www.lisha.ufsc.br/

Abstract: This article presents C-MAC, a *Configurable Protocol* for medium access control in wireless sensor networks. C-MAC works as a framework of medium access control strategies, with a transparent configuration system. The protocol aggregates different services, each implemented in several different strategies. Applications may configure different communication parameters in compile-time and run-time. C-MAC's meta-programmed implementation yields smaller footprint and higher performance than equivalent protocols for wireless sensor networks.

1. INTRODUCTION

Narrow band radios typically support simple modulation schemes, leaving a high level of control to software, and thus paying a higher processing overhead. On the other hand, wide band radios support sophisticated modulation techniques, such as Direct-Sequence Spread Spectrum (DSSS), and Phase Shift Keying (PSK), which are more robust to noise and interference, but have small flexibility and high power consumption. Low power radios have thus been presented as a viable alternative for wireless communication for embedded systems such as sensor networks.

Given the simplicity of communication hardware for such systems, Medium Access Control protocols and other data link layer services must be implemented in software. Services such as data packet detection, error detection and treatment, addressing, packet filtering, and others traditionally implemented in hardware become one of the main parts of

Wanner, L-F., de Oliveira, A.B., Fröhlich, A.A, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 401–410.

a communication stack implemented by operating systems for wireless sensor networks.

In current hardware prototypes for sensor networks, transmitting a single byte of data through the radio consumes more power than executing hundreds of instructions in the main micro-controller (Langendoen and Halkes, 2005). Thus, a communication system for wireless sensor networks must use the communication resources in a conservative way, providing only the support needed for specific applications to implement their services.

On the other hand, the simplicity of communication hardware for wireless sensor networks is not a limiting factor, but a desirable characteristic, as it allows a wide range of configuration of the data communication channel. Previous studies (Langendoen and Halkes, 2005; Polastre et al., 2005) show medium access control implemented in software which are adequately designed and adapted to applications may be more efficient than standardized, hardware-implemented protocols.

Several protocols have been designed and implemented for medium access control in wireless sensor networks (El-Hoiydi, 2002; Hoesel and Havinga, 2004; Lu et al., 2004; Woo and Culler, 2001; Mahlknecht and Boeck, 2004; Ringwald and Roemer, 2005; Dam and Langendoen, 2003; Polastre et al., 2004; Ye et al., 2002). In the context of wireless sensor networks, where different applications may have very specific communication workloads, adaptability and configuration are key issues. However, several implementations for these protocols in real systems do not provide adequate mechanisms for the application to configure the communication channel. In fact, many protocols, such as S-MAC (Ye et al., 2002) are specifically designed to optimize predetermined workloads (such as multi-hop communications), and present little to nil opportunities for application-guided configuration.

This article presents the design and implementation of C-MAC, a *Configurable Protocol* for medium access control in wireless sensor networks. C-MAC works as a framework of medium access control strategies, with a transparent configuration system. The protocol aggregates different services (e.g. synchronization, data detection, acknowledgments, contention, sending and receiving), each implemented in several different strategies. Applications may configure different communication parameters in compile-time and run-time. C-MAC owes its fundamental concepts to different medium access control protocols, and presents a new project to well-known, efficient low-power wireless communication solutions, allowing applications to take advantages of strategies that fit their requirements. Section 2 discusses MAC protocols for wireless sensor networks. Section 3 presents the design and implementation of

C-MAC. Section 4 evaluates the presented solution, and Section 5 closes the paper.

2. MAC PROTOCOLS FOR WIRELESS SENSOR NETWORKS

A Medium Access Control (MAC) protocol decides when a network node may access the medium, and tries to ensure that one node doesn't interfere with the transmissions of a second node. The MAC protocol is also responsible of treating or signaling collisions to the upper layers in the protocol stack.

Most MAC protocols for wireless networks have been designed and optimized for satellite links and local wireless networks (*Wireless LANs*). Communication requirements for wireless sensors are considerably different from these scenarios, specially regarding the need of autonomous and energy efficient operation in a wireless sensor. In several application scenarios of wireless sensors, parameters such as low latency and high throughput have smaller importance (Ali et al., 2006).

In current hardware prototypes for sensor networks, transmitting a single byte of data through the radio consumes more power than executing hundreds of instructions in the main micro-controller. Thus, power efficient operation is the main design issue to be considered for MAC protocol design. Langendoen and Halkes, 2005 identify the main sources of power overhead in a MAC protocol for wireless sensors with small traffic:

Idle listening: If a node doesn't know when it will receive messages from one of its neighbors, it will have to keep the radio turned on in receive mode all the time. As the cost of reception is much greater than the cost of *standing-by*, this is perhaps the greatest source of overhead.

Collisions: If two nodes transmit at the same date, data are corrupted, both transmissions must be repeated, and the energy spent on the first tries is wasted.

Overhearing: As the radio channel is a shared medium, a receiver may hear packets which are not directed to itself.

Traffic fluctuations: When a phenomenon is detected by many neighboring nodes in a densely installed network (in the *local gossip* pattern), the nodes will compete for the radio channel, and will waste power while waiting for a transmission window.

Medium access control protocols for sensor networks compromise performance (latency, throughput) for cost (power consumption). Power consumption is minimized mainly by shortening the period in which the radio listens to the channel when there are no communications (*idle listening*) (Langendoen and Halkes, 2005).

Contention-based protocols, such as B-MAC (Polastre et al., 2004) attain energy efficiency by increasing the message preamble, allowing the radio channel to be verified with lower periodicity. Slot-based protocols, such as S-MAC (Ye et al., 2002) and T-MAC (Dam and Langendoen, 2003), reduce power consumption by limiting communication to well-defined periods. Comparisons in different application scenarios show that there is no "optimal" protocol for wireless sensors (El-Hoiydi, 2002; Hoesel and Havinga, 2004; Lu et al., 2004; Woo and Culler, 2001; Mahlknecht and Boeck, 2004; Ringwald and Roemer, 2005; Dam and Langendoen, 2003; Polastre et al., 2004; Ye et al., 2002). The choice of an adequate MAC protocol for a wireless sensor network application depends on the level of compromise between power efficiency and communication flexibility. Characteristics such as complexity, special hardware requirements (e.g. synchronization hardware), and application data communication patterns must be taken into consideration when determining the ideal MAC for a given scenario. In what regards communication support in an operating system for wireless sensors applications, configuration flexibility may be considered the most desirable trait.

3. C-MAC

C-MAC is a *Configurable Protocol* for medium access control in wireless sensor networks equipped with low power radio transceivers. Its configurable characteristic allows the user to adjust several communication parameters (e.g. synchronization, data detection, acknowledgments, contention, sending and receiving), in order to adjust the protocol to the needs of different applications.

Santos and Fröhlich, 2005 have developed a static composition system of lightweight protocols for computer grids that has presented several advantages to monolithic layered based implementations, such as the Internet Protocol (TCP/IP). This system consists of a meta-programmed framework, which provides mechanisms to allow the selection, configuration, and combination of communication protocols according to application requirements; and a baseline architecture for communication on top of which the protocols are designed. This paradigm presents several advantages, including the ability to create new communication services on-demand, and to allow applications to experiment with dif-

ferent communication protocol configurations, by collecting metrics in order to identify the configuration best fitted to its requirements.

C-MAC uses these same premises to build a configurable communication kernel, over which other protocols may be composed. The list of configurable points in the C-MAC architecture was assembled by analyzing the commonalities and variabilities in the design of MAC protocols for wireless sensor networks, and aims at delivering the largest range of configurable points that, when combined, form a complete MAC protocol. Protocol configuration is performed at compile-time, and run-time configuration of protocol characteristics is not treated in the current C-MAC architecture. The overhead of maintaining several configuration possibilities programmed in the node, and the need of a second protocol for synchronization makes the use of a run-time configuration system impracticable for a protocol as widely configurable as C-MAC. The main C-MAC configuration points include:

Basic Communication Characteristics: These configurations are handled by the communication hardware, and include: transmission frequency and power (which may be altered in runtime); modulation type (e.g. Manchester, NRZ); transmission data rate.

Duty cycle and organization: The duty cycle determines the active period in which the radio may operate. In a simple CSMA-based configuration, the radio may transmit at any time it detects the channel is free. On the other hand, in a slot-based protocol, the duty cycle is limited to the active part of the protocol's time slot.

Collision-avoidance mechanism: The collision-avoidance mechanism in a wireless sensor networks MAC protocol may be comprised of a carrier sense algorithm, the exchange of contention packets (*Request to Send* (RTS) and *Clear to Send* (CTS)), or a combination of both. Furthermore, there must be the possibility to not use any collision-avoidance mechanism, for example, in a sparse network with little communication, in which eventually retransmitting corrupted packets is less costly than the mechanism itself.

Collision-detection mechanism: As hardware for communications in wireless sensor networks is mostly half-duplex, the most widely used mechanism for collision detection is the use of acknowledgment packets, sent from the receiving node to indicate that the data was correctly received. In situations where packet loss is not a problem (e.g. a densely installed network, where many information packets are redundant), the collision detection mechanism

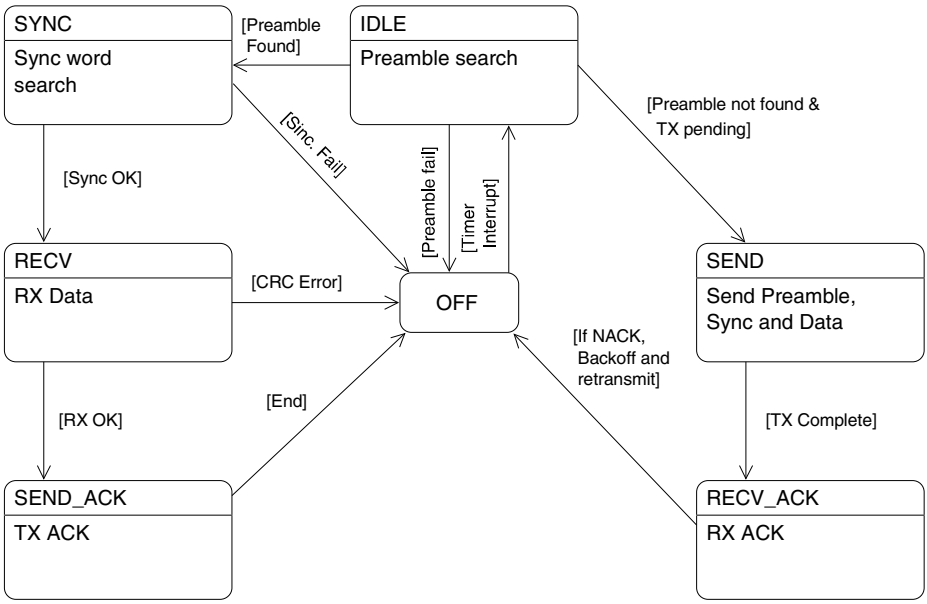


Figure 1. C-MAC State Machine Overview

may be eliminated from the protocol configuration, thus increasing power efficiency.

Collision handling mechanism: When a collision is detected, the protocol may retransmit the packet, or simply increment a packet loss counter.

C-MAC is implemented through a state machine in which transitions are activated through interruptions of a dedicated counter (which controls duty cycle and backoffs), and through data interruptions of the communication hardware. Figure 1 presents a simplified overview of the protocol's main state machine.

C-MAC's configurable characteristics are selected by the programmer through *Configurable Traits* in EPOS (Polpeta and Fröhlich, 2004; Fröhlich, 2001), an application-oriented operating system. The EPOS system framework allows software components to be automatically adapted to fulfill the requirements of particular applications. Configurable Traits are parametrized classes whose static members describe the properties of a certain class. When a certain property is selected, the functionality it describes is included into the protocol. On the other hand, due to the use of function inlining and static meta-programming when a certain characteristic is *not* selected, no overhead associated with it is added to the final object code of the protocol. Furthermore,

Table 1. Communication parameters

Parameter	Value
Duty cycle	100%
Transmission power	5 dBm
Modulation	19.2 kbps, Manchester encoding
Transmission backoff	0 ms
Collision detection and handling	None
Maximum theoretical data rate	16.4 kbps

Table 2. Memory footprint

System	Code (bytes)	Data (bytes)
EPOS	3888	108
TinyOS	8562	205

C-MAC's modular design allows different radio transceivers to be used with no alterations in the protocol's logic.

4. EVALUATION

In order to allow comparison between C-MAC in EPOS and other MAC protocols, C-MAC was configured in order to function like B-MAC in the TinyOS system. Table 1 summarizes the configuration used both in C-MAC and B-MAC in the tests presented in this section. Table 2 presents the system footprint for both TinyOS and EPOS. C-MAC's meta-programmed implementation, along with EPOS's component architecture delivers equivalent functionality, yielding a considerably smaller footprint than the B-MAC protocol in TinyOS.

Figure 2 presents packet loss in a network of two to four nodes, using both systems. In this test, a node was positioned in the center of an imaginary circle with a two-meter radius, and the other nodes were positioned in the border of this circle, equidistant form each other. Each node is in range, and may potentially interfere in the communications of every other node. Figure 3 shows the throughput of this same network. The differences between the two equally configured protocols may be attributed to a series of factors, including slight variations in the test environment and heterogeneity between nodes (e.g. antennas, slight variations in battery charge); design and programming problems; loss of interruptions; and the overhead of internal system procedures.

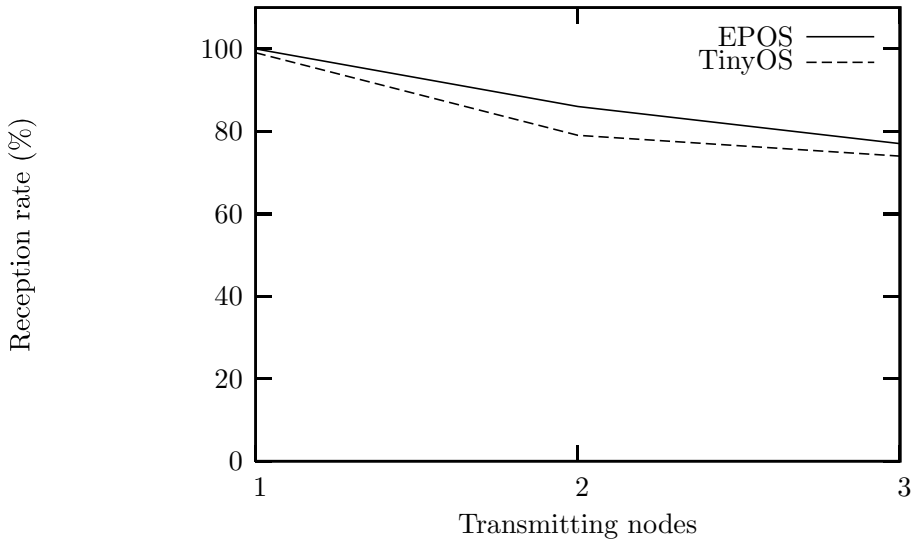


Figure 2. Reception rate (B-MAC and C-MAC)

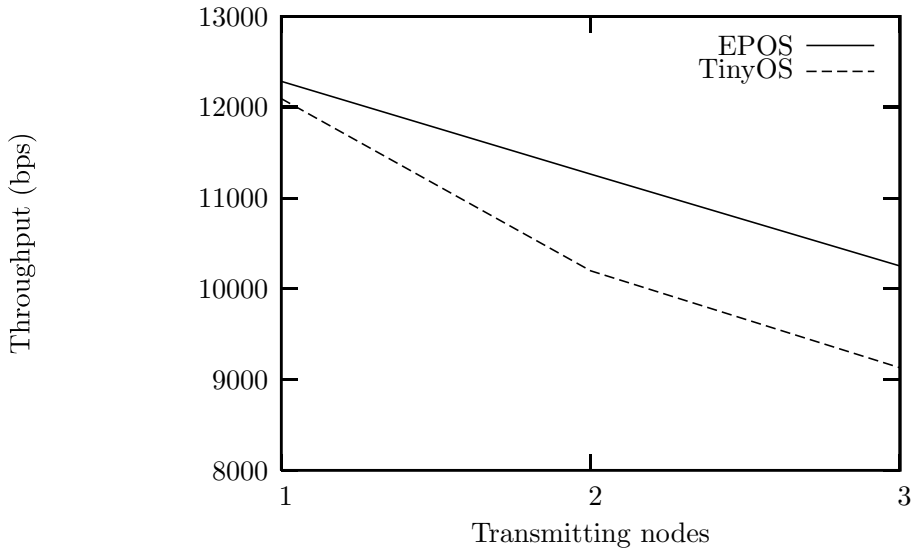


Figure 3. Throughput (B-MAC and C-MAC)

It should be noted that repetitions in the tests diminish the impact of environmental variations and node heterogeneity.

C-MAC presented slightly superior performance than a protocol configured in an equivalent fashion, with smaller memory footprint. This advantage is magnified by C-MAC's configuration system, which allows the creating of application-specific protocols, with only the necessary overhead.

5. CONCLUSION

The choice of an adequate MAC protocol for a wireless sensor network application depends on the level of compromise between power efficiency and communication flexibility. In what regards communication support in an operating system for wireless sensors applications, configuration flexibility may be considered the most desirable trait. The implementation of the B-MAC protocol for the TinyOS system allows dynamic and static configuration of basic communication parameters (e.g. frequency, power). Alterations in protocol logic demand direct interference of the user in the source code. In the same system, the S-MAC protocol implementation uses a monolithic design, with little space for configuration.

C-MAC's modular project and implementation allows applications to configure the data communication channel according to its needs. The protocol aggregates different services (e.g. synchronization, data detection, acknowledgments, contention, sending and receiving), each implemented in several different strategies. Applications may configuration different communication parameters in compile-time and run-time. Tests using C-MAC in the EPOS operating system yielded better results than the B-MAC protocol in TinyOS, when both protocols were configured in an identical fashion.

Future developments in the C-MAC protocol shall allow a greater level of configuration including the possibility of global network synchronization, and adaptative duty cycle.

REFERENCES

Ali, Muneeb, Saif, Umar, Dunkels, Adam, Voigt, Thiemo, Romer, Kay, Langendoen, Koen, Polastre, Joseph, and Uzmi, Zartash Afzal (2006). Medium access control issues in sensor networks. *SIGCOMM Comput. Commun. Rev.*, 36(2):33–36.

Dam, T. van and Langendoen, K. (2003). An adaptive energy-efficient MAC protocol for wireless sensor networks. pages 171–180, Los Angeles, CA.

El-Hoiydi, A. (2002). Aloha with preamble sampling for sporadic traffic in ad hoc wireless sensor networks. In *IEEE International Conference on Communications (ICC)*, New York.

Fröhlich, Antônio Augusto (2001). *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin.

Hoesel, L. van and Havinga, P. (2004). A lightweight medium access protocol (LMAC) for wireless sensor networks. Tokyo, Japan.

Langendoen, K. and Halkes, G. (2005). *Embedded Systems Handbook*, chapter Energy-Efficient Medium Access Control. CRC press.

Lu, G., Krishnamachari, B., and Raghavendra, C. (2004). An adaptive energy-efficient and low-latency MAC for data gathering in sensor networks. In *Int. Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN)*, Santa Fe, NM.

Mahlknecht, S. and Boeck, M. (2004). CSMA-MPS: A minimum preamble sampling MAC protocol for low power wireless sensor networks. In *IEEE Int. Workshop on Factory Communication Systems*, pages 73–80, Vienna, Austria.

Polastre, Joseph, Hill, Jason, and Culler, David (2004). Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA. ACM Press.

Polastre, Joseph, Szewczyk, Robert, and Culler, David (2005). Telos: Enabling ultra-low power wireless research. In *The Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, Los Angeles, California.

Polpeta, Fauze Valério and Fröhlich, Antônio Augusto (2004). Hardware Mediators: a Portability Artifact for Component-Based Systems. In *International Conference on Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 271–280, Aizu, Japan. Springer.

Ringwald, M. and Roemer, Kay (2005). BitMAC: a deterministic, collision-free, and robust MAC protocol for sensor networks. In *Proc. IEEE European Workshop on Wireless Sensor Networks (EWSN) 2005*, pages 57–69, Istanbul, Turkey.

Santos, Thiago Robert and Fröhlich, Antônio Augusto (2005). A Customizable Component for Low-Level Communication Software. In *19th Annual Symposium on High Performance Computing Systems and Applications*, pages 58–64, Guelph, Canada.

Woo, Alec and Culler, David E. (2001). A transmission control scheme for media access in sensor networks. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 221–235, Rome, Italy.

Ye, W., Heidemann, J., and Estrin, D. (2002). An energy-efficient MAC protocol for wireless sensor networks. In *21st Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1567–1576.

INTEGRATING WIRELESS SENSOR NETWORKS AND THE GRID THROUGH POP-C++

Augusto B. de Oliveira[1], Lucas F. Wanner[1], Pierre Kuonen[2] and Antônio A. Fröhlich[1]

[1] *Laboratory for Software and Hardware Integration*
Federal University of Santa Catarina
PO Box 476 – 88049-900 – Florianópolis, SC, Brazil
{*augusto, lucas, guto*}*@lisha.ufsc.br*

[2] *Grid and Ubiquitous Computing Group*
University of Applied Sciences of Fribourg
PO Box 32 – CH-1705 – Fribourg, FR, Switzerland
pierre.kuonen@eif.ch

Abstract: The topic of interaction between Wireless Sensor Networks (WSNs) and other computation systems has received relatively low scientific attention, and the interface between the data source and the applications that use that data remains a problem for the application programmer. This work extends POP-C++, a programming language and runtime support system for Grid programming, to enable Grid applications to seamlessly and concurrently use WSNs' sensing and processing capabilities.

Keywords: WSN, Grid, Remote Objects

1. INTRODUCTION

Even though Wireless Sensor Networks (WSNs) have been the focus of many research effors in recent years, the topic of interaction of WSNs with other computing systems has received relatively low attention. The research efforts that do address this issue, such as TinyDB and Cougar, abstract the individual sensor nodes and give access to the WSN as a whole, allowing the applications to perform queries as they would to a database; while this level of abstraction allows for the optimization of queries, minimizing the amount of messages to be sent over the wireless link, we feel that it takes power away from the application programmer, as further exploration of the WSN nodes' processing capabilities becomes

de Oliveira, A.B., Wanner, L.F., Kuonen, P., Fröhlich, A.A, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 411–420.

difficult. Furthermore, such a solution does not hide the frontier between the WSN and the rest of the computing system.

In contrast to these approaches, we integrate WSNs and the Grid seamlessly without removing power from the application programmer by extending POP-C++ [Nguyen and Kuonen, 2007]. POP-C++ is a pre-existing object-oriented grid programming language and runtime support system, capable of supporting distributed, parallel objects over a network. The specific goals of our extension are to allow:

- Grid applications to communicate with WSNs seamlessly: By hiding all network interaction under a remote method call interface, details of the network stack and physical medium are transparent to the application;
- Concurrent use of the WSN sensing capabilities by multiple Grid applications: By allowing multiple objects to run on each node and each object to be used by multiple interfaces, concurrent use of the WSN by multiple applications is made possible;
- Application independent sensor node software: by allowing applications to use a common set of methods, we hope to minimize the need for costly re-programming of node program memory and allow additional applications to be initiated after the WSN is deployed.

The structure of this article is the following: In section 2 we introduce the POP-C++ programming language and runtime support system; in section 3 we detail how POP-C++ was extended into WSNs; in section 4 we evaluate our implementation; in section 5 we present the design challenges found in WSNs; section 6 presents related works that share our goal; finally, in section 7 we present our conclusions.

2. POP-C++

POP-C++ is an extension of C++ created to support requirement driven, distributed parallel objects. In POP-C++'s object model, parallel objects have the ability to describe their resource needs at runtime and are allocated in any of the remote nodes that can support its execution. The process of finding a suitable node and transmitting the object code is transparent to the programmer. POP-C++ also has special method invocation semantics, but syntatically the method invocation statements do not differ between local and remote invocations. Furthermore, parallel objects are shareable, that is, references to an object can be passed in any method, be it local or remote.

The POP-C++ runtime architecture consists of three actual objects for each parallel class the user implements: the Interface, the Broker and

the actual Object. The Interface is an object itself, instantiated in the caller side; it shares the method interface of the actual Object, giving the transparency of interaction for the application.

The Broker is the callee-side correspondent to the Interface, it receives method calls from the network, unpacks the data, calls the method on the actual Object and then repacks the return value and sends it back to the Interface. The actual Object is the implementation of the user, with the code that is to be distributed.

POP-C++ introduces two syntax extensions to C++ in addition to the declaration of parallel classes: Requirement descriptions and Method Semantics.

- Requirement descriptions: Using an associated object description, the developer can express resource requirements in the form of a hostname, the number of MFlops needed, the amount of memory needed and the communication bandwidth needed between itself and its interfaces.

- Method invocation semantics: The invocation semantic options are defined at compile time by the application programmer and can be classified in two types, Interface-side and Object-side:

  - Interface side semantics can be either Synchronous or Asynchronous; they control at which time the Interface-side method returns. In Synchronous mode, the caller waits until the method on the Object returns; this is analogous to traditional method invocation. Asynchronous methods return immediately, allowing the caller to continue execution.
  - Object side semantics can be either Mutex, Sequential or Concurrent. Mutex semantics guarantee no concurrency on the object, Sequencial semantics guarantee no concurrency on the particular method it is applied to, and Concurrent semantics allow full multi-threaded execution.

3. EXTENDING POP-C++ INTO WSNS

To give the system architect an uniform model with which to program grid applications that use WSNs, we extended the POP-C++ model to WSNs. That means that not only should the programmer be able to instantiate Interfaces in sensor nodes to Objects running in other nodes, but also instantiate Interfaces to those Objects from inside the Grid. There should be no difference between "normal" Grid-to-Grid remote method calls and those performed from the Grid to the WSN.

```
parclass SensorNode
{
    public:
        SensorNode(int node, string machine) @{ od.url(machine);};

        async seq void setLEDs(char val);
        sync conc int getTemperature();

};
```

Figure 1. Basic SensorNode POP-C++ Class

Figure 1 illustrates the implementation and instantiation of an Object that runs on the WSN and receives function calls from the Grid. The implementation has methods to read temperature sensor values and set a value to be displayed on the LEDs of the node. Any node on the Grid may instantiate an Interface to this object and transparently call methods to it.

3.1 Compromises

Due to the low-resource nature of WSN nodes, the WSN implementation of the POP-C++ runtime support system had to occupy as little program and main memory as possible. This has caused us to make some compromises in the implementation:

- No interchangeable communication protocols: When in a Grid environment, it is not only interesting but necessary to support multiple, interchangeable communication protocols. When in WSNs, though, the communication protocol over the wireless link is very likely to be constant and global. Therefore, our implementation does not support multiple communication protocols at runtime.
- No dynamic resource allocation: While in Grid nodes the process of dynamic resource allocation is just a matter of downloading and executing a binary, re-writing program memory on WSN nodes is a very costly procedure, in terms of energy [Dunkels et al., 2006]. To diminish the need for re-programming, the system architect is encouraged to provide low level functions in addition to his applications' high-level routines; if a problem is found on the high-level code or a new application is to be run on the Grid, similar functionality can be attained from the aggregation of lower level calls.
- Limited Parallelism: Because of the very small amount of main memory available on sensor nodes, the amount of concurrent

threads that can run on a node is also very small. This means a limited amount of method calls, regardless of semantic, will be able to execute in real concurrency, and that the following incoming calls will have to be queued.

3.2 Addressing

To allow direct access to each individual sensor node, we had to extend the addressing method for POP-C++ objects. There is the possibility for POP-C++ Interfaces to be instantiated with a hostname parameter, forcing the Object to be allocated in that machine. We expanded this method to the sensor nodes, requiring two addresses:

- Address 1 - Point of contact between Grid and WSN: All WSNs must have at least one point of contact to the Grid. This point of contact must be able to communicate in both the protocol used by the Grid and the protocol used by the WSN, so it will most likely require special hardware such as the radio transceiver found in the sensor nodes. By taking this parameter we are able to instantiate a Proxy Broker in the appropriate Grid node, creating the logical bridge that forwards the method calls directed at the WSN.
- Address 2 - WSN Node: This address enables the Proxy Broker to direct the method calls to the correct sensor node. Its format is left open because different addressing methods can be used inside different WSNs.

3.3 The Proxy Broker

To allow the method calls to be forwarded into the WSN, a special Broker object has been created. This is a generic Broker that simply receives method calls from the Grid as if it was the Object's real broker, then forwards them to the WSN node. Once the WSN node returns from the method call, this Broker forwards the return value to the original caller. This creates the effect of transparency to the Interfaces of that Object; to them, the method calls are never leaving the Grid.

Figure 2 shows the Grid connected to the WSN through the Proxy Brokers; note that there may be more than one point of contact between the Grid and each WSN.

4. EVALUATION OF POP-C++ OVER WSNS

In this section we evaluate the overhead that our POP-C++ runtime system introduces by comparing two implementations of the following application: getting and setting an 8-bit value, that is to be displayed

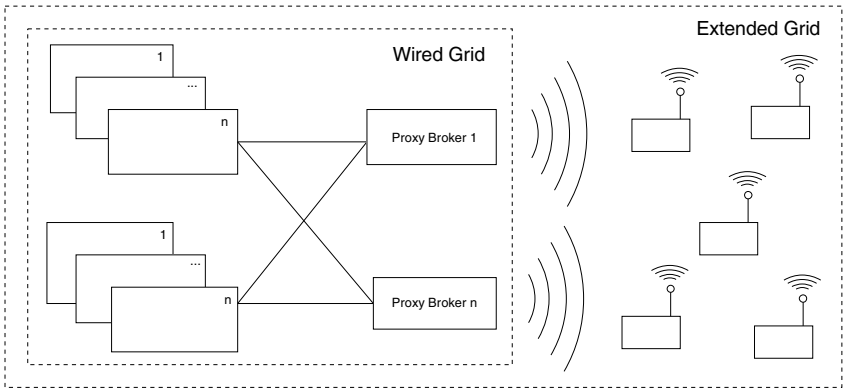


Figure 2. Proxy Brokers integrating the wired Grid and the WSN nodes

at the node's LEDs. One version was implemented over POP-C++ and the other directly on top of the operating system. In the case of the POP-C++ implementation, the client instantiates an Interface to a "SensorNode" Object that is executed on the other node, and calls the `get()` and `set()` methods to retrieve and set the data. In the native implementation, the client sends pre-formatted packets that are opened by the server and replied to with the data as payload.

4.1 Hardware testbed

The tests were made using two Mica2 sensor nodes developed at Berkeley; they use a single-channel CC1000 radio, an 8MHz Atmel Atmega128 8-bit microcontroller, 4KB of main memory and 128KB of program flash memory.

4.2 Runtime support

To provide the communication, memory management and concurrency support that both applications need we used the Embedded Parallel Operating System [Fröhlich and Schröder-Preikschat, 1999] (EPOS). It consists in a component-based framework for generating runtime support for dedicated computing applications. For this test EPOS' MAC protocol was configured for reliability, ensuring packet delivery through acknowledgements and minimizing network delays through an always-on duty cycle.

4.3 Benchmarks

- Packet size: Figure 3 details the network packet size and content for both implementations of this aplication; POP-C++ packets are larger for 2 reasons:
  - Object Field: To allow for more than one Object to be executed in each node, packets are individually addressed;
  - Semantic Value Field: The semantic values of the method to be called are also represented in the header.

  The addition of equivalent functionality on the native implementation would result in a similar packet size, which is justified by the additional information that must be transferred when supporting a complex set of applications.

- Grid-Sensor Requests-per-Second: To evaluate the overall overhead of the POP-C++ runtime system on this application we conducted a performance test that measured the Requests-per-Second that could be made from the client node to the server node. Figure 4 shows that the POP-C++ implementation could perform 6.875 remote method calls per second, and the native implementation was able to perform 7.046 requests per second. This difference of 2.42% is due to the additional processing performed by the POP-C++ runtime system as method calls arrive from the network.

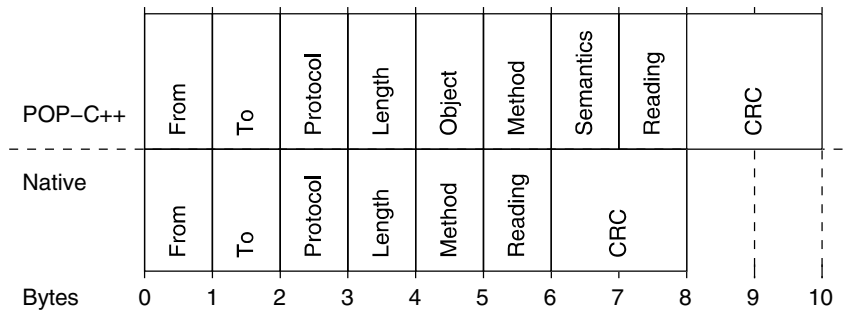


Figure 3. Packet Size Comparison

5. DESIGN CHALLENGES

In this section we discuss the new issues brought by the environment POP-C++ would now work in.

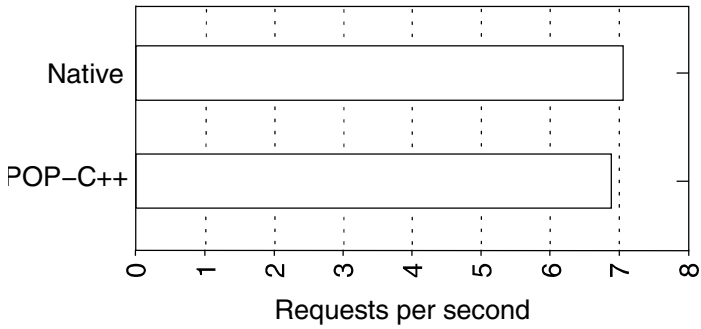


Figure 4. Requests-per-Second Comparison

5.1 Network Load

- Payload Data Overhead: To transport the additional fields necessary to make the function call, we include them in the header of our packets. This is an overhead that the original POP-C++ system also has, but it has a greater negative effect in the WSN implementation because of the high energy cost of transmitting data over the wireless link. To minimize this issue we reduced the size of each of these fields from the original 32 bit values, expecting sensor nodes to host a smaller amount of objects and methods.

- Packet Exchange Overhead: While the original POP-C++ system used additional ACKs, in the WSN implementation we leave any error correction or flow control up to the network stack of the operating system; this way, the medium access and routing protocols can handle communication in the way they see fit.

5.2 Scalability

If all method calls from the Grid to the WSN are routed through a single Grid node, failure of that node would cause all Objects running on the sensor network to become incommunicable. Under very heavy loads it may also become a network bottleneck, causing heavy contention at the physical level. To circumvent that, we allow multiple, concurrent points of contact between the Grid and the WSN, ideally physically separated so all can transmit concurrently.

5.3 Security/QoS

The issues of Security and QoS in WSNs are still relatively unexplored, but current research on these areas propose solutions at the network

level, specially at the routing layer. There are groups working on methods to protect WSNs against DoS attacks [Deng et al., 2005], guarantee packet confidentiality [Banerjee and Mukhopadhyay, 2006] and provide QoS [Ouferhat and Mellouck, 2006] on a shared WSN. POP-C++ uses the network stack of its underlying operating system as an application would, and in a similar way to the Grid implementation, we rely on EPOS to provide this kind of functionality.

6. RELATED WORK

In this section we briefly discuss other projects of note that share our goal of allowing the Grid to communicate with WSNs, and discuss how their approach relates to ours.

TinyDB [Madden et al., 2003], Cougar [Yao and Gehrke, 2002] and other research efforts [Madden et al., 2002] [Bonnet et al., 2001] implement distributed query processors, putting great effort into query optimization and efficient routing. Using these techniques they have achieved considerable reduction in power consumption in addition to externalizing a more friendly SQL-like interface to the application programmer. Our extension of POP-C++ does share all these goals, but instead of active optimization, it gets out of the way of the application programmer allowing full access to the sensor nodes' hardware. Also, we see the possibility of implementing the query optimization features of TinyDB and Cougar as POP-C++ object code, yielding similar functionality.

Hourglass [Gaynor et al., 2004] inserts a Data-Collection Network (DCN) between the application and the sensor networks they acquire data from. Hourglass abstracts the internals of the sensor networks completely, and provides traditional functionality such as service registration and discovery, as well as routing the data from the sensor networks to the applications. Hourglass' approach is very internet-oriented, and uses several estabilished standards such as XML, SOAP and OGSA. In this scheme, our POP-C++ extension could be used behind the Sensor Entry Point to perform all communication in the WSNs and provide a data stream to the DCN.

7. CONCLUSION

In this article we describe a way to use remote parallel objects to integrate the Grid and WSNs, by extending the POP-C++ runtime system into the sensor network. We believe that by using POP-C++ to perform this integration we enable the application programmer to

use the WSN for multiple applications transparently, by using locally instantiated interfaces to objects that run on the sensor nodes.

When comparing a functionally equivalent application implemented with and without POP-C++, our runtime system showed a small overhead cost that was justified by the ability to support multiple applications concurrently.

REFERENCES

Banerjee, S. and Mukhopadhyay, D. (2006). Symmetric key based authenticated querying in wireless sensor networks. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 22, New York, NY, USA. ACM Press.

Bonnet, P., Gehrke, J., and Seshadri, P. (2001). Towards sensor database systems. In *MDM '01: Proceedings of the Second International Conference on Mobile Data Management*, pages 3–14, London, UK. Springer-Verlag.

Deng, J., Han, R., and Mishra, S. (2005). Defending against path-based dos attacks in wireless sensor networks. In *SASN '05: Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks*, pages 89–96, New York, NY, USA. ACM Press.

Dunkels, A., Finne, N., Eriksson, J., and Voigt, T. (2006). Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 15–28, New York, NY, USA. ACM Press.

Fröhlich, A. A. and Schröder-Preikschat, W. (1999). EPOS: an Object-Oriented Operating System. In *2nd ECOOP Workshop on Object-Orientation and Operating Systems*, volume CSR-99-04 of *Chemnitzer Informatik-Berichte*, pages 38–43, Lisbon, Portugal.

Gaynor, M., Moulton, S. L., Welsh, M., LaCombe, E., Rowan, A., and Wynne, J. (2004). Integrating wireless sensor networks with the grid. *IEEE Internet Computing*, 8(4):32–39.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2002). Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2003). The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA. ACM Press.

Nguyen, T.-A. and Kuonen, P. (2007). Programming the grid with pop-c++. In *Future Generation Computer Systems (FGCS)*, volume 23. N.H. Elsevier.

Ouferhat, N. and Mellouck, A. (2006). Qos dynamic routing for wireless sensor networks. In *Q2SWinet '06: Proceedings of the 2nd ACM international workshop on Quality of service & security for wireless and mobile networks*, pages 45–50, New York, NY, USA. ACM Press.

Yao, Y. and Gehrke, J. (2002). The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18.

MODELING OF SOFTWARE-HARDWARE COMPLEXES

Panel

K. H. (Kane) Kim (Organizer)
DREAM Lab., EECS Dept.
University of California, Irvine, CA, 92697-2625 USA
khkim@uci.edu

Various issues related to modeling of software-hardware complexes, including the following, will be addressed.

- Are those currently available modeling approaches insufficient for use in systematic design and optimization of embedded software + hardware systems?
- Are there potential synergies between software modeling and hardware modeling approaches?
- What are the possibilities of and obstacles in combining some software modeling approaches and some hardware modeling approaches?
- What are the kinds of things that software-modeling experts wish to learn from the work on hardware modeling?

The following researchers will serve as panelists:

- Nikil Dutt, UCI, USA
- Hermann Kopetz, Vienna Univ. of Technology, Austria
- Franz Rammig, Univ. of Paderborn, Germany
- Wayne Wolf, Princeton Univ., USA
- Kane Kim, UCI, USA

Position statements of these panelists are in the following.

(Kane) Kim, K.H., 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 421–421.

MODELING OF SOFTWARE-HARDWARE COMPLEXES

Position Statement

Nikil Dutt
ACES Laboratory, Center for Embedded Computer Systems.
Donald Bren School of Information and Computer Sciences.
University of California, Irvine, CA 92697-3435, USA.
dutt@uci.edu

Complex embedded systems typically comprise a large number of interconnected software and hardware components operating in highly dynamic environments that need to deliver user-specified QoS under demanding design constraints: performance, energy consumption, system cost, system reliability, etc. As the software content of such embedded systems continues to increase, it would appear that the boundary between reasoning about hardware versus software begins to blur. However, modeling strategies for such software-hardware complexes need to *holistically* embody key characteristics of both hardware and software domains in order to effectively capture various facets of design, analysis, verification and synthesis. It becomes even more important in the context of effective co-design of such software-hardware complexes. This is true for the entire gamut of embedded systems that range in size and complexity from "small" embedded Systems-on-Chip (SoCs) to large "system-of-systems" that comprimise multiple, distributed, heterogeneous embedded systems working in consort to deliver a desired set of services for the end user.

Such complex embedded systems have several distinguishing characteristics, including, but not limited to:

- *Complexity and scale*. The distributed nature of emerging embedded systems poses significant challenges for managing the complexity of interactions among distributed components, and managing the computational and network resources to keep the system coordinated.

Dutt, N, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 423–425.

- *Dynamic and uncertain operating environments*. Embedded systems often operate in dynamic and uncertain environments due to a variety of reasons, including unpredictable user requests, hardware and software resource failures, environmental noise, and incomplete knowledge of the system operating state.
- *Stringent timing requirements*. Timing plays a critical role in the correct performance of any embedded system. Indeed, timing is part of key functionality for mission-critical applications that must execute tasks reliably within stringent deadlines.
- *Energy awareness*. Modern embedded systems (regardless of their scale and complexity) increasingly need to address energy as a first-class design constraint.
- *Cross-Layer interactions*. Complex embedded systems are highly networked, and involve end-to-end interactions among multiple layers (application, middleware, network, OS, hardware architecture) in a distributed environment. Thus there is a need for a holistic modeling mechanism that captures such cross-layer interactions.

To operate such systems effectively while maintaining the desired QoS, multiple performance-related parameters must be dynamically tuned to adapt to changing application modes and operating conditions. In addition to performance management, the system's operational constraints and timing characteristics must be verified at design time to ensure its correctness, feasibility, and safety. To verify correct operation, the designer must make necessary (and sometimes strong) assumptions about the services provided by the underlying execution platform, such as guaranteed task execution rates and deadlines, reliable message communication between system components, etc. These requirements must therefore be satisfied by the execution platform (i.e., the middleware, operating systems, and networks) to ensure that the management framework performs correctly at runtime.

Model-based techniques have recently been proposed as a promising approach for capturing, managing and refining overall system behavior in a reliable and efficient manner. Domain specific models can be generated to match the idiosyncrasies of specific application domains, while a generic model-based framework can provide a backbone to address a variety of problems (e.g., power management, efficient resource allocation and provisioning) using well-established concepts and techniques.

The software and hardware communities have much to give, as well as to learn from each other. A holistic integration of best practices and proven

technologies from each domain is only the start; we will need to develop modeling techniques that are simultaneously domain-aware and platform-compliant, in order to tame the ever increasing complexity and challenges posed by tomorrow's software-hardware complexes.

ENHANCING A REAL-TIME DISTRIBUTED COMPUTING COMPONENT MODEL THROUGH CROSS-FERTILIZATION

Position Statement

K. H. (Kane) Kim
DREAM Lab., EECS Dept.
University of California, Irvine, CA, 92697-2625 USA
khkim@uci.edu

The need for overall optimization of software-hardware complexes has been there throughout the history of computer applications. However, the need for significant improvement in the techniques for achieving it has become very acute as the growth of the embedded computing application field has been in an accelerating mode since mid-1990's. As a natural consequence of it, the desire to have unified modeling approaches that are effectively applicable to both hardware systems and software systems is as strong as ever.

Often software design activities and hardware design activities proceed largely independently without joint analysis and optimization, i.e., without disturbing each other, until the integration phase is reached. In such situations, software designers can benefit considerably from the availability of reliable models of hardware under development at the early stage. Similarly, hardware designers can benefit from the availability of early abstract but reliable models of software under development. Both designers are bound to wonder about the essential differences between hardware system modeling techniques and software system modeling techniques and about the aspects common to both types of modeling techniques.

This author, who is a researcher dealing with techniques for software system modeling, feels that the following complimentary relationship exists between the recent work on software system modeling and that on hardware system modeling.

- Software designs used to involve producing multiple layers of abstraction. In general, the number of layers defined during software

(Kane) Kim, K.H, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 427–430.

design has been significantly larger than that defined during hardware design. Therefore, in developing software modeling techniques, the ability for handling multiple layers of abstraction has been emphasized.

- On the other hand, the software modeling research has involved timing specifications at coarse levels in comparison to the hardware modeling research. Therefore, it seems worthwhile for software modeling researchers to attempt to learn from the techniques developed for timing specifications as parts of hardware models and use that knowledge to enhance software modeling techniques.
- In addition, the software modeling research has involved concurrency exploitation and specifications at coarse levels in comparison to the hardware modeling research. Again, it seems worthwhile for software modeling researchers to attempt to learn from the techniques developed for concurrency specifications as parts of hardware models and use that knowledge to enhance software-modeling techniques without damaging the strong ability to deal with multiple layers of abstraction.

One of the major challenges that researchers in software system modeling have faced is to establish techniques effective in modeling of real-time networked computing software. In particular, the modeling techniques that help the developers in *safety check*, i.e., analyzing software designs to check about the possibility of violating action timing requirements inherent in the given applications, have been wanted. Research in this important area has been advancing rather slowly.

The ease of such safety analysis depends heavily on the structure of the real-time networked computing software. Yet, the art of structuring real-time networked computing software that eases such safety check while enabling efficient and flexible design of high-performance software has remained immature. Here several potentially conflicting desiderata exist. First of all, the structuring approach must enable maximal exploitation of concurrency since otherwise, it will lead to designs of low-performance software of which inability to meet certain stringent timing requirements in certain applications is quite obvious. Yet, careless exploitation of concurrency leads to designs that are hard to analyze. Secondly, timing specification must be done in terms that can be supported by the execution engine consisting of hardware, operating system kernel, and middleware. Without defining and realizing new-generation execution engines, the timing specifications are bound to be in low-level terms and it is hard to check whether such specifications lead to efficient safe overall designs or not.

This author and his collaborators have been enhancing the structuring scheme called the *Time-triggered Message-triggered Object* (TMO) scheme as well as associated execution engine models and software engineering

tools in the past 15 years (Kim 1997, Kim 2000, Kim et al. 2005). TMO is a programming model of real-time distributed computing software components. The TMO scheme is intended to enable maximal exploitation of concurrency while maintaining a high degree of analyzability of the real-time distributed computing software. At present the TMO research community is seriously interested in learning from the hardware modeling research community about additional potential mechanisms for concurrency and timing specification.

Attempts have been made from the beginning to incorporate into TMO a practically sufficient set of mechanisms for expressing timing constraints without violating the fundamental modular structuring principle underlying the object / component structuring schemes. So far, all reported experiments and experiences seem to indicate that the set of mechanisms is practically sufficient but it is too early to conclude as such.

The essence of the modeling power of the TMO scheme is as follows (details in Kim 1997, Kim 2000, Kim et al. 2005):

- *Active components*, i.e., components which have hearts or internal energy sources and thus may be of hardware type or software + physical_ or virtual_hardware type, can be modeled as TMOs because of the availability of the *time-triggered method*, also called the *spontaneous method* (SpM) mechanism in TMO.
- Interconnections among components can be modeled by logical multicast channels called *Real-time Multicast and Memory-replication Channels* (RMMCs) or *service requests* which are sent from TMOs to TMOs and may be of one-way communication type, two-way non-blocking communication type, or two-way blocking communication type.
- All real time references in TMO are references to *global time* that is commonly accessible from all distributed computing sites and may be maintained in a decentralized fashion. Action timings of components can thus be specified in terms of global time. Clock-driven actions and periodic actions can be specified as parts of the time-triggered method specifications in TMO.
- Signal delays incurred over interconnection links can be represented by use of *official release times* (ORTs) of messages (RMMC messages or service requests) or combinations of production time-stamps and ORTs.

Again, what can be learned from hardware modeling techniques to further enhance the TMO scheme toward supporting more efficient types of networked real-time embedded computing systems is a question that we plan to address more intensively in coming years.

REFERENCES

Kim, K.H., 1997, Object structures for real-time systems and simulators, *IEEE Computer*, **30**(8): 62 – 70.

Kim, K.H., 2000, APIs for real-time distributed object programming, *IEEE Computer*, **33**(6): 72 – 80.

Kim, K.H., Li, Y., Liu, S., Kim, M.H., Kim, D.H., 2005, RMMC programming model and support execution engine in the TMO programming scheme, *in Proc. of the 8th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*, pp. 34–43, Seattle, USA.

MODELING OF SOFTWARE-HARDWARE COMPLEXES

Position Statement

Hermann Kopetz
Technische Universität, Wien, Austria
hk@vmars.tuwien.ac.at

The ever-increasing complexity level of embedded systems, the technology trends of the semiconductor industry to large production series of chips, the new possibilities of the FPGA technology to develop customized hardware of substantial size by software techniques, and the increased competition in the world market entail the need for an integrated development strategy for embedded hardware-software systems.

From a strictly functional point of view, where a user accesses the services of a component, it is irrelevant whether the services of the component are provided by software that is executed on a commercial off-the-shelf computer or by a special hardware-solution (e.g., by a dedicated field-programmable gate array--FPGA). From the non-functional points of view, there are substantial differences: For example, the power efficiency of an FPGA solution is two orders of magnitude better than the power efficiency of a software-on-CPU solution. If we go to a custom hardware solution (i.e., a specially designed chip), the power-efficiency can be improved by another order of magnitude. Power-efficiency is a very important system property in battery-operated devices. Similar differences can be noticed concerning speed and needed silicon area of the implementation of a given function.

These observations suggest that a uniform modeling approach for hardware-software systems is appropriate for the design and implementation of embedded systems. At the Platform-Independent-Model (PIM) level the services of a component should be specified without regard to the chosen implementation platform. Since embedded systems are time-sensitive, this PIM specification must include the temporal properties as well as the value properties of the services. Here we can learn from hardware modeling, since

Kopetz, H, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 431–432.

a state-for-the-art hardware specification contains detailed information about a variety of temporal parameters: clock speed, delays, jitter, just to name a few.

We feel that a message-based view is an appropriate level of abstraction for the service specification of a component at the PIM level. A message is an atomic unit that combines the data aspects (value domain) and the timing aspects (instant when a message is sent and instant when a message is received) into a single concept. The emerging field of WEB services provides a good starting point for the specification of component services using this message abstraction. While the message specification in the value domain and in the domain of message exchange patterns are well covered, a precise temporal specification of the message transmission and message arrival (that is needed in embedded systems) is not part of the WEB service specification. WEB services are intended to be used in an open environment (the WEB environment), where precise temporal properties are difficult to guarantee. At present, the precise specification of the temporal properties of message-based services at the PIM level is still an open research issue.

Given that such a precise specification of the message-based component services (in the value and time domain) is available, it is a question of compiler technology to translate this specification into a platform specific model (PSM) for the chosen execution platform, i.e, into a form that can be executed on a CPU (classical software solution) or in a form that can be directly executed on the target hardware (e.g., on an FPGA). Such a model-based approach that distinguishing clearly between the PIM and the PSM has a number of distinct advantages: a given PIM design can be implemented in different implementation technologies (PSMs) without changing the service specification of the component and without a need to retest the system that uses the modified component. For example a new application can be implemented at first in software-on-a-CPU, and the follow up implementation for the mass market can be implemented in an FPGA or a dedicated ASIC chip. Furthermore the issue of technology obsolescence is addressed: if, during the lifetime of the embedding system (e.g., an airplane) the original hardware chips are no longer available, a new version of the service can be implemented on a new hardware base without changing the service specification of the component.

SOFTWARE-HARDWARE COMPLEXES: TOWARDS FLEXIBLE BORDERS

Position Statement

Franz J. Rammig
Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany
franz@upb.de

For a long time it was relative clear what the terms hardware and software are standing for. Hardware was just the platform for the execution of software. This idea of a platform later was adapted by system software as well. From this point of view the idea of platform-driven design did emerge. The previously used term "Meet in the Middle", created by Hugo de Man describes this idea very clearly. It is a mixed bottom-up and top-down design method, meeting in the middle. Given some design target concurrently two design activities take place. One (bottom-up) tries to develop a fixed platform that is adapted to the specific needs of the application. "Adapted" means in most cases that a broader class of platforms is just customized towards the specific needs. At the same time an initial abstract solution of the problem is stepwise refined until it can be directly executed on the customized platform.

This approach today is widely accepted, at least in academic environments. There are even formal methods available to support such an approach like formal refinement calculi like the B-method or various process algebraic techniques like π-calculus.

Modelling such complexes can follow traditional approaches. OO techniques, e.g. UML did prove to be well suited to support such platform-oriented design methods. Big effort has been spent to study UML in the context of system engineering including hardware specific aspects. During a design cycle which follows the MDA principles hardware parts can be refined to more hardware oriented modelling languages like System-C or System-Verilog. Looking at any kind of platform just as a service providing entity unifies this design process. What has to be designed for the "Meet in

Rammig, F.J, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 433–435.

the Middle" approach is just a proper application programming interface (API). Whether such an API is given by a set of primitives (as in the case of operating systems, see Posix for example) or an instruction set is a minor distinction from the point of view modelling principles. The term "Transaction Level Simulation (TLS)" used in modern hardware design makes this unification of basic principles even more evident. Simple instructions are abstracted to more complex services provided by the hardware.

So it seems that we did achieve a certain agreement how to model mixed hardware software complexes. The situation changes, however, as soon as we allow dynamic reconfiguration at run-time.

If so we damage the cornerstone of platform-oriented design. There is no longer a clear distinction what is platform and what is application. Assume a micro-programmable processor where the micro-program can be altered during run-time. UCI's NISC (No Instruction Set Computer) approach can be seen as an attempt into this direction. Any kind of dynamically reconfigurable hardware (FPGAs or similar devices) may serve as another example. Again, the set of services offered may be altered at run-time.

A platform in such a scenario still is a virtual entity where applications can be deployed on. But now it becomes highly dynamic what set of services is offered, where such services are located, in which way these services are provided and implemented. Even the distinction between platform and applications becomes dynamic. In essence this means that now decisions that usually are made at design-time are deferred to run-time, i.e. have to be made by the system itself and automatically.

An approach to model such situations may be based on the assumption that both, the application software and the hierarchy of virtual platforms will be organized in a strict component-based way. This means that we assume a fabric of components with well-defined interfaces in between. Both, the application programs and the execution platform follow a reflective paradigm. They collect knowledge about their own state and these ones of objects they are connected with.

An application tries to require as little fixed private resources as possible (it virtually may have to pay for any kind of resources). Therefore it is interested in getting rid of as many components as possible by handing over them to the platform and just paying for using the services.

A platform is interested to collect as many components as possible as then it may charge usage fees to applications. On the other hand it has to pay for its private usage of resources as well. So a benefit exists for a platform only if it owns components that it can offer to a variety of applications. This model means that components that can be shared functionally between many applications should have a tendency to be handed over to the platform while

dedicated components will remain part of the respective applications. This makes evident that in this model a platform is rarely stable. It will adapt itself dynamically to requests by dynamically changing applications just by inclusion and exclusion of services. It will do the same internally, i.e. internally it becomes an application that treats lower levels of services as its platform.

This completely dynamic model of hardware-software complexes is fundamentally different from traditional approaches. Novel means of modelling, analysis, and synthesis will be requested to handle such systems in an adequate manner. It is the strong belief of the author that this kind of "dynamization" will happen, it already started to happen. So the question is not whether we have to deal with such situations but how to. The research community is requested to investigate potential approaches.

EMBEDDED SW DESIGN SPACE EXPLORATION AND AUTOMATION USING UML-BASED TOOLS

Flávio R. Wagner and Luigi Carro
Computer Science Institute, Federal University of Rio Grande do Sul (UFRGS), Brazil
{flavio, carro}@inf.ufrgs.br

Abstract: This tutorial discusses design space exploration and software automation based on an UML front-end. First, we review software automation tools targeted at the embedded systems domain. Following, we present an approach for the estimation of memory, performance, and energy of a given application modeled from an initial UML specification. We proceed with an analysis of the possibilities of linking different modeling environments for software generation (Simulink and UML, for example). Finally, we show the possibilities of using other specification languages to obtain more abstraction and allow design space exploration together with software automation.

Key words: Embedded Software, Design Space Exploration, High-level modeling, UML, MDA, code generation

1. INTRODUCTION

The increasing complexity of embedded systems design, which is derived from the amount of functionality that is required from these systems, together with the shortening of the life cycle of embedded products, results in a design scenario where productivity and quality are simultaneously required in order to deliver competitive products.

Selic[1] emphasizes that the use of techniques starting from higher abstraction levels is crucial to the design success. The UML language has gained in popularity as a tool for specification and design of embedded systems. There are many efforts that describe the use of UML during the different phases of an embedded system design [2].

Wagner, F.R., Carro, L, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 437–440.

It is widely known that design decisions taken at higher abstraction levels can lead to substantially superior improvements. This context suggests the support to a fast design space exploration in the early design steps. However, software engineers, when developing an application using UML, do not have a concrete measure of the impact of their modeling decisions on issues such as performance and energy for a specific embedded platform.

On the other hand, the specification of typical embedded systems requires several models of computation, and in a single product (like a cell phone) many of these models can be simultaneously found, like event-driven, sequential code, and synchronous dataflow. This pushes the need for supporting software automation under different models of computation, a task not completely supported by any current software automation tool.

This tutorial discusses two critical aspects of embedded systems software automation, namely the design space exploration (DSE) from high-level UML models, and automation techniques to serve several models of computation. The tutorial is organized in the topics that follow.

1) Review of current software automation tools

Most of the academic and commercial solutions for software automation focus on the management of huge domain-specific systems, because conventional software is usually suited for a single domain. Some tools auotmate code generation for any application specified in a proper way, such as Unimod[3], which extracts code from UML diagrams and follows the Model Driven Architecture[4] approach. However, tools that automate general and conventional software development are not aware of code optimizations for memory or power, a crucial step for embedded systems because of their tight restrictions.

More recent approaches targeted to embedded software development[5] keep working on extensions of state machines, in an Esterel[6] fashion, and try to improve final code optimizations for control-dominated reactive systems. The Model-Integrated Computing (MIC) approach fully adopts the model-based development paradigm[7]. As the approach relies on domain-specific modeling languages[8], it is also not suited for applications that mix different models of computation.

The Ptolemy framework[9] provides simulation and prototyping for heterogeneous embedded systems. Although Ptolemy provides abstraction for many domains, it requires knowledge of several different languages, and this may increase development time. As a matter of fact, none of the presented approaches targets the ultimate goal of providing high abstraction to increase software production, with the necessary design space exploration to meet embedded systems' tight constraints.

2) UML modeling and high-level design space exploration

UML modeling solutions for the same application may result in very different physical costs, as demonstrated by Oliveira[10]. We propose the estimation of physical costs such as performance, energy, and memory footprint directly from UML models. We show how these estimations may be integrated into an automatic DSE process, which is guided by optimization heuristics (such as simulated annealing or ant colony optimization), following desired design goal and constraints.

3) Software synthesis using an MDA-based approach

In the context of platform-based design, software synthesis is mainly based on the reuse of a library of previously designed components. For a rapid DSE process, it is essential that a designer can quickly generate software for different mappings from a specification of the application into the platform. This can be accomplished by an MDA-based approach, where the application and the platform are specified according to appropriate meta-models and the mapping between them is implemented as a sequence of model transformations. We present a DSE tool that can automatically optimize the mapping of PIM (Platform Independent Model) into PSM (Platform Specific Model). From this mapping, and according to an implementation meta-model, the final software may be generated.

Our approach uses a meta-modeling infrastructure proposed by Nascimento[11], which is based on OMG standards, such as MOF and XMI. Different languages for modeling the application, the platform, and the final implementation may be used, given the existence of appropriate translators from these languages into instances of the generic meta-models. This approach, in fact, may be used also for the synthesis of dedicated hardware modules, in a HW-SW co-design methodology. We support UML and Simulink because recent efforts[2,12,13] show that both languages are considered attractive for System Level design.

4) Software modeling and synthesis for different models of computation

A comparison between UML and Simulink modeling approaches concluded that both have pros and cons[14]. UML provides all benefits from the object-oriented paradigm, like modularity, encapsulation, reusability, etc. In the other side, Simulink supports multiple models of computation and complete code generation. To address multiprocessor systems, a Simulink-based software synthesis approach was recently developed[15], which starts from a Combined Architecture Application Model (CAAM) described in Simulink and generates multithread code. It includes optimization techniques to generate memory-efficient[15] and communication-efficient code[16].

However, building the CAAM model using the Simulink GUI can be error-prone, and usually software engineers prefer to employ UML. Then, Brisolara[17] proposes mapping rules to translate a UML model into a Simulink CAAM, thus allowing the use of UML as a front-end for the Simulink multithread code generation flow. It allows the integration of both modeling languages in a unique design flow, which combines the benefits of UML and Simulink.

REFERENCES

[1] B.Selic. Models, Software Models and UML. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, 2003. Chapter 1, p. 1-16.
[2] L.Lavagno, G.Martin, B.Selic. UML for Real: Design of Embedded Real-Time Systems. Kluwer Academic Publishers, 2003.
[3] Executable UML Unimod. v.1.3, 2003. http://unimod.sourceforge.net.
[4] OMG. Model Driven Architecture. White Paper v. 3.2, November 2000.
[5] F.Balarin et al. Synthesis of Software Programs for Embedded Control Applications. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, v. 18, n.6, June 1999.
[6] G.Berry et al. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of Computer Programming, v. 19, n. 2, 1992.
[7] J. Sztipanovits et al. Model-Integrated computing. IEEE Computer. April 1997.
[8] G.Karsai et al. Model-Integrated Development of Embedded Software. Proceedings of the IEEE, v. 91. n. 1, January 2003.
[9] J.Buck et al. Ptolemy: a framework for simulating and prototyping heterogeneous systems. International Journal in Computer Simulation, v. 4, 1992.
[10] M.Oliveira et al. Early Embedded Software Design Space Exploration Using UML-based Estimations. RSP'06, Chania, Greece. June 2006.
[11] F.A.Nascimento et al. ModES: Embedded Systems Design Methodology and Tools based on MDE. MOMPES 2007, Braga, Portugal, March 2007.
[12] SysML: Systems Modeling Language. http://www.omgsysml.org/. July 2006.
[13] R.Boldt. Combining the Power of MathWorks Simulink and Telelogic UML/SysML-based Rhapsody to Redefine the Model-Driven Development Experience. Telelogic White Paper, June 2006. http://www.ilogix.com/whitepaper-overview.aspx.
[14] L. Brisolara et al. A Comparison between UML and Function Blocks for Heterogeneous SoC Design and ASIP Generation. In: G.Martin and W.Mueller (Ed.). UML for SoC Design. Chapter 9. Springer, 2005.
[15] K. Huang et al. Simulink-Based MPSoC Design Flow: Case Study of Motion-JPEG and H.264. DAC'07, San Diego, USA, June 2007.
[16] L.Brisolara et al. Reducing Fine-grain Communication Overhead in Multithread Code Generation for Heterogeneous MPSoC. SCOPES'07. Nice, April 2007.
[17] L.Brisolara et al. Using UML as a front-end for an efficient Simulink-based multithread code generation targeting MPSoCs. UML-SoC'07, San Diego, USA, June 2007.

MEDICAL EMBEDDED SYSTEMS

Roozbeh Jafari,[1] Soheil Ghiasi,[2] and Majid Sarrafzadeh[3]

[1]*Department of Electrical Engineering*
University of Texas at Dallas
rjafari@utdallas.edu

[2]*Department of Electrical and Computer Engineering*
University of California, Davis
soheil@ece.davis.edu

[3]*Department of Computer Science*
University of California, Los Angeles
majid@cs.ucla.edu

Abstract: Light-weight embedded systems for medical monitoring are often referred to low-profile, small size, unobtrusive and potable processing elements with limited power resources. Such systems typically incorporate sensing, processing and communications and are often manufactured to be simple and cost-effective. Being low profile and wearable immediately implies the limitations in computational capabilities, memory (storage), speed and I/O interfaces. In this tutorial, we portray a brief description of low-power and light-weight embedded systems; present two pilot applications; and illustrate their corresponding design challenges. We specifically discuss the reconfiguration technqiues.

1. INTRODUCTION

Light-weight embedded systems are often referred to low-profile, small size, unobtrusive and potable processing elements with limited power resources. Such systems typically incorporate sensing, processing and communications and are often manufactured to be simple and cost-effective. Being low profile immediately implies the limitations in computational capabilities, memory (storage), speed and I/O interfaces. Despite their low complexity impedes the handling of computationally intensive tasks, they may be deployed in collaborative networks and potentially in large quantities. Their sensing capabilities allow their seamless integration with the physical world whilst their general-

Jafari, R., Ghiasi, S., Sarrafzadeh, M, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 441–444.

purpose architecture designs yield notable advantages such as reconfigurability and adaptability with various applications and environments. Consumers, on the other hand, constantly demand thinner, smaller and lighter systems with smaller batteries in which the battery life is enhanced to meet their lifestyle. Improving the performance of battery life, however, has been always a major scientific challenge for researchers. Several other key factors such as reliability, security, customizability and reconfigurability has been among the major concerns.

2. TARGET APPLICATIONS

2.1 Physical Movement Monitoring

Health care costs in developed countries are rapidly increasing due to a substantial increase the elderly population. Monitoring of daily physical activities can be a key to evaluating the actual quality of life among the elderly. We believe that the overall health and wellness of elderly sectors of the population can greatly benefit from the use of information and communication technology, especially for the homebound. Sensor platforms integrated into clothing provide the possibility of enhanced reliability of accident reporting and health monitoring. Such devices improve the independence of people needing living assistance [1]. In this tutorial, we outline a framework for movement assessment and classifying physical activities. In particular, we are interested in classifying transition movements for example, sit-to-stand, stand-to-sit, lie-to-stand and stand-to-lie etc. The functionality of our automated pattern recognition system is divided into three basic tasks: the preprocessing and filtering, the description task which generates attributes of a movement using feature extraction techniques, and the classification task which classify this movement based on its attributes.

2.2 Electrocardiogram Analysis

The electrocardiogram (ECG) is a record of variation of bioelectric potential with respect to time as the human heart beats. Due to its ease of use and non-invasiveness, ECG plays an important role in patient monitoring and diagnosis. Multi-channel electrocardiogram (ECG) data provide cardiologists with essential information to diagnose heart disease in a patient. Our primary objective is to address the feasibility verification of implementing an ambulatory ECG analysis algorithm with real-time diagnosis functions for wearable computers. ECG analysis algorithms have always been very difficult tasks in the realization of computer aided ECG diagnosis. Implementation of such algorithms becomes even harder for small and mobile embedded systems that should meet the given latency requirements while minimizing overall energy

dissipation for the system [2]. These obstacles may dramatically reduce the effectiveness of embedded distributed algorithms. Thus, a new distributed, embedded, computing attribute, dynamically reconfigurable, must be developed and provided to such systems. In these systems, reconfiguration capability, in particular, may be of great advantage. This capability can adaptively alter the system configuration to accommodate the objectives and meet the constraints for highly dynamic systems. Similar to movement monitoring, ECG analysis consists of preprocessing, pattern recognition, and classification.

3. APPLICATION CHARACTERISTICS

Most medical monitoring applications rely on pattern recognition and classification. Given the goal of classifying objects based on their attributes, the functionality of an automated pattern recognition system can be divided into two basic tasks: The description task generates attributes of an object using feature extraction techniques, and the classification task assigns a group label to the object based on the attributes with a classifier. There are two different approaches for implementing a pattern recognition system: statistical and structural. Each approach utilizes different schemes within the description and classification tasks which incorporates a pattern recognition system. Statistical pattern recognition [3][4] concludes from statistical decision theory to discriminate among data from different groups based upon quantitative features of the data. The quantitative nature of statistical pattern recognition, however, makes it difficult to discriminate among groups based on the morphological (i.e., shape-based or structural) sub-patterns and their interrelationships embedded within the data. This limitation provided the impetus for development of structural approaches to pattern recognition.

4. RECONFIGURATION DESIGN TECHNIQUES

Wearable health monitoring applications are characterized by their highly dynamic behavior, constrained hardware platform, critical missions, and potentially very large share in the marketplace. Consequently, the design process for such applications needs to be rethought to consider their specific requirements and platform challenges. Adaptive resource assignment and execution is one of the key requirements of medical applications. In this section, we present a number of situations that call for adaptive execution of target applications on medical embedded systems. We highlight several key research problems that will be discussed at length in our tutorial. These problems have to be addressed in order to develop adaptive resource assignment services.

4.1 High-Level Application Restructuring

A unique characteristic of our target architecture is its ability in re-capturing sufficiently-correlated physical data from different body sensors. For example, heart beat signal can be sensed at a number of different locations on body. Therefore, a high-level application running on sufficiently sensor-populated wearable system that processes heart beat signal, has several choices for reading in the signal [2]. We illustrate high-level transformations that restructure a given application to take advantage of this specific application-dependent portunity. Example transformations include input re-capturing and task duplication. Besides input recapturing for sufficiently-correlated input data, replicating selected internal computations can create a favorable tradeoff between communication and computation demand, which is a great incentive to restructure the application at runtime.

4.2 Dynamic Code Migration

While some runtime events require new tasks to be invoked, another group of runtime events can be handled by partial restructuring of the application via task reallocation, without creation of new computations. For example, a newly launched application can overload a resource that is already executing a number of applications. In that case, we might be able to migrate some computing tasks, as well as the associated communications, to other resources to address the problem. Similarly, we can handle transient faults by reassignment of affected computations to other resources at price of degrading system performance, or increased energy dissipation.

REFERENCES

[1] Roozbeh Jafari, Wenchao Li, Ruzena Bajcsy, Steven Glaser and Shankar Sastry. "Physical Activity Monitoring for Assisted Living at Home". In *International Workshop on Wearable and Implantable Body Sensor Networks (BSN)*, March 2007.

[2] Roozbeh Jafari, Hyduke Noshadi, Soheil Ghiasi and Majid Sarrafzadeh. "Adaptive Electrocardiogram Feature Extraction on Distributed Embedded Systems". *IEEE Transactions on Parallel and Distributed Systems Special Issue on High Performance Computational Biology*, 17(8):797–807, Aug 2006.

[3] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley and Sons, Inc., second edition, 2000.

[4] Anil K. Jain, Robert P. W. Duin, and Jianchang Mao. Statistical pattern recognition: A review. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(1):4–37, 2000.