Babak Falsafi
T. N. Vijaykumar (Eds.)

LNCS 3164

Power - Aware Computer Systems

Third International Workshop, PACS 2003
San Diego, CA, USA, December 2003
Revised Papers

Springer

Lecture Notes in Computer Science 3164

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Babak Falsafi T.N. Vijaykumar (Eds.)

Power - Aware Computer Systems

Third International Workshop, PACS 2003
San Diego, CA, USA, December 1, 2003
Revised Papers

Springer

Volume Editors

Babak Falsafi
Carnegie Mellon University
Electrical and Computer Engineering, Computer Science
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
E-mail: babak@cmu.edu

T.N. Vijaykumar
Purdue University
School of Electrical and Computer Engineering, Department of Computer Science
465 Northwestern Avenue, West Lafayette, Indiana 47907-1285, USA
E-mail: vijay@ecn.purdue.edu

Library of Congress Control Number: Applied for

CR Subject Classification (1998): B.7, B.8, C.1, C.2, C.3, C.4, D.4

ISSN 0302-9743
ISBN 3-540-24031-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11364641 06/3142 5 4 3 2 1 0

Preface

Welcome to the proceedings of the 3rd Power-Aware Computer Systems (PACS 2003) Workshop held in conjunction with the 36th Annual International Symposium on Microarchitecture (MICRO-36). The increase in power and energy dissipation in computer systems has begun to limit performance and has also resulted in higher cost and lower reliability. The increase also implies reduced battery life in portable systems. Because of the magnitude of the problem, all levels of computer systems, including circuits, architectures, and software, are being employed to address power and energy issues. PACS 2003 was the third workshop in its series to explore power- and energy-awareness at all levels of computer systems and brought together experts from academia and industry.

These proceedings include 14 research papers, selected from 43 submissions, spanning a wide spectrum of areas in power-aware systems. We have grouped the papers into the following categories: (1) compilers, (2) embedded systems, (3) microarchitectures, and (4) cache and memory systems.

The first paper on compiler techniques proposes pointer reuse analysis that is biased by runtime information (i.e., the targets of pointers are determined based on the likelihood of their occurrence at runtime) to map accesses to energy-efficient memory access paths (e.g., avoid tag match). Another paper proposes compiling multiple programs together so that disk accesses across the programs can be synchronized to achieve longer sleep times in disks than if the programs are optimized separately.

The first paper on embedded systems proposes scaling down the components (display, wireless, and CPU) of a mobile system to match user requirements while reducing energy. The second paper explores an integer linear programming approach for embedded systems to decide which instructions should be held in a low-power scratchpad instead of a high-power instruction cache. The next paper predicts battery life at runtime to help the operating system in managing power. The next paper proposes a tiled architecture that exploits parallelism enabled by global interconnects and synchronized design to achieve high energy efficiency. The last paper in this group proposes a policy to decide which of the multiple wireless network interfaces provided in a mobile device should be used based on the power and performance needs of the mobile system.

The third group of papers focuses on microarchitecture techniques, and includes an analysis of energy, area, and speed trade-offs between table lookup for instruction reuse and actual computation. Another paper proposes scheduling transactions in a multiprocessor to as few CPUs as possible to increase the number of CPUs in deep-sleep state. The next paper evaluates the extent of energy savings achieved by avoiding instructions that are either not needed for correct behavior or not committed, and by sizing microarchitectural structures. The last paper proposes coupled power and thermal simulation and studies the effect of temperature on leakage energy.

The last group proposes techniques to reduce power in caches and memory. The first paper in this group studies the interaction between dynamic voltage scaling (DVS) and power-aware memories and proposes policies to control the CPU's DVS setting and the memory's power setting together. The next paper uses the criticality of instructions to determine which locations should be placed in high-speed cache banks and which in low-power banks. The last paper proposes applying high-performance techniques only to the most-frequently-used instruction traces and saving power on the other traces.

PACS 2003 was successful due to the quality of the submissions, the efforts of the program committee, and the attendees. We would like to thank Pradip Bose for his interesting keynote address, which described microarchitectural choices at the early architecture-definition stage to achieve power and energy efficiency. We would like to also thank Glen Reinman, Jason Fritts, and the other members of the MICRO-36 organizing committee who helped arrange the local accommodations and publicize the workshop.

December 2003 Babak Falsafi and T.N. Vijaykumar

PACS 2003 Program Committee

Babak Falsafi, *Carnegie Mellon University (co-chair)*
T.N. Vijaykumar, *Purdue University (co-chair)*

Sarita Adve, *University of Illinois*
David Albonesi, *University of Rochester*
David Blaauw, *University of Michigan*
Pradip Bose, *IBM*
David Brooks, *Harvard University*
George Cai, *Intel*
Keith Farkas, *Hewlett–Packard*
Yung-Hsiang Lu, *Purdue University*
Mahmut Kandemir, *Pennsylvania State University*
Ulrich Kremer, *Rutgers University*
Diana Marculescu, *Carnegie Mellon University*
Andreas Moshovos, *University of Toronto*
Farid Najm, *University of Toronto*
Daniel Mosse, *University of Pittsburgh*
Raj Rajkumar, *Carnegie Mellon University*
Hazim Shafi, *IBM*
Josep Torrelas, *University of Illinois*
Amin Vahdat, *Duke University*

Table of Contents

Compilers

Embedded Systems

Microarchitectural Techniques

Cache and Memory Systems

Runtime Biased Pointer Reuse Analysis and Its Application to Energy Efficiency

Yao Guo, Saurabh Chheda, and Csaba Andras Moritz

Department of Electrical and Computer Engineering,
University of Massachusetts, Amherst, MA 01003
{yaoguo, schheda, andras}@ecs.umass.edu

Abstract. Compiler-enabled memory systems have been successful in reducing chip energy consumption. A major challenge lies in their applicability in the context of complex pointer-intensive programs. State-of-the-art high precision pointer analysis techniques have limitations when applied to such programs, and therefore have restricted use. This paper describes runtime biased pointer reuse analysis to capture the behavior of pointers in programs of arbitrary complexity. The proposed technique is runtime biased and speculative in the sense that the possible targets for each pointer access are statically predicted based on the likelihood of their occurrence at runtime, rather than conservative static analysis alone. This idea implemented as a flow-sensitive dataflow analysis enables high precision in capturing pointer behavior, reduces complexity, and extends the approach to arbitrary programs. Besides memory accesses with good reuse/locality, the technique identifies irregular accesses that typically result in energy and performance penalties when managed statically. The approach is validated in the context of a compiler managed memory system targeting energy efficiency. On a suite of pointer-intensive benchmarks, the techniques increase the fraction of memory accesses that can be mapped statically to energy efficient memory access paths by 7-72%, giving a 4-31% additional L1 data cache energy reduction.

1 Introduction

The memory system, including caches, consumes a significant fraction of the total system power. For example, the caches and translation look-aside buffers (TLB) combined consume 23% of the total power in the Alpha 21264 [7], and the caches alone use 42% of the power in the StrongARM 110 [8]. Recent studies have proposed compiler-enabled cache designs [2, 12, 14] to improve cache performance as well as energy consumption. A major challenge, however, is their applicability when dealing with complex pointer-intensive programs. This paper presents a new approach to deal with complex pointer-intensive programs in such schemes based on the idea of runtime biased pointer reuse analysis. In addition to compiler-enabled memory systems, applications such as compiler-based prefetching, software-based memory dependence speculation, and parallelization, could also significantly benefit from the techniques presented in this paper.

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 1–12, 2004.

Many researchers have focused on program locality/reuse analysis for array-based memory accesses [9, 15, 16]. In general, array accesses are more regular than pointer-based memory accesses because arrays are normally accessed sequentially while pointers typically have more complicated behavior. Array based accesses are also relatively easy to deal with as type information is available to guide the analysis.

Intensive use of pointers makes however program analysis difficult since a pointer may point to different locations during execution time; the set of all locations a pointer can access at runtime is typically referred to as the *location set*. This difficulty is further accentuated in the context of large and/or complex programs. For example, more precise dataflow-based implementations of pointer analysis have limitations (e.g., often cannot complete analysis) when used for large programs or when special constructs such as pointer based calls, recursion, or library calls are found in the program. The less precise alias analysis techniques (e.g., those that are flow-insensitive) have lower complexities but don't provide precise enough static information about pointer location sets.

Our objective is to develop new techniques to capture pointer behavior that can be used to analyze complex applications with no restrictions, while providing good precision. The idea is to determine pointer behavior by capturing the frequent locations for each pointer rather than all the locations as conservative analysis would do. Predicted pointer reuse is therefore runtime biased and speculative in the sense that the possible targets for each pointer access are statically predicted/speculated based on the likelihood of their occurrence at run-time. The approach enables lower complexity and possibly higher precision analysis than traditional dataflow based approaches because locations predicted to be infrequently accessed are not considered as possible targets. The approach is applicable in all architecture optimizations that use some kind of compiler-exposed speculation hardware and when absolute correctness of static information leveraged is not necessary. This includes for example compiler managed energy-aware memory systems, compiler managed prefetching, and speculative parallelization and synchronization - these applications by their design would benefit from precise memory behavior information and would tolerate occasional incorrect static control information.

This paper shows the application of the proposed pointer techniques to an energy-efficient compiler-enabled memory management system published previously, called Cool-Mem [2]. The Cool-Mem architecture achieves energy reduction by implementing energy efficient statically managed access paths in addition to the conventional ones. The compiler decides which path is used based on static information extracted. For accesses that reuse the same cache line, cache mapping information is maintained to help eliminate redundancy in cache disambiguation. Whenever the compiler can correctly channel data memory accesses to the static access path, significant energy reduction is achieved; the statically managed access path does not need Tag access and associative lookup in RAM-Tag caches, and Tag access in CAM-Tag caches. We show that the Cool-Mem

architecture, if extended with our techniques, is able to handle pointer based accesses and achieve up to 30% additional energy savings in the L1 data cache.

The rest of this paper is structured as follows. Section 2 presents the runtime biased compiler analysis techniques, including pointer analysis, distance analysis, and reuse analysis algorithms. Following this, Sect. 3 provides an overview of the compiler-enabled memory framework used for simulation and Sect. 4 shows the experimental framework. Finally, Sect. 5 gives the experimental results gathered through simulation, and we conclude with Sect. 6.

2 Compiler Analysis

The runtime biased (RB) pointer reuse analysis can be separated into a series of three steps: RB pointer analysis, RB distance analysis, and RB reuse analysis.

RB Pointer Analysis is first applied in order to gather basic pointer information needed to predict pointer access patterns. A flow-sensitive dataflow scheme is used in our implementation. Flow-sensitive analysis maintains high precision (i.e., the location set of each pointer access is determined in a flow-sensitive manner even if based on the same variable). Our analysis is guided by reevaluating, at each pointer dereference point, the (likely) runtime frequency of each location a pointer can point to. For example, possible locations that are from definitions in outer loop-nests are marked or not included when the pointer is dereferenced in inner loops and if at least one new location has been defined in the inner loop. Conventional analysis would not distinguish between these locations.

Precise conventional pointer analysis usually requires that the program includes all its source codes, for all the procedures, including static libraries. Otherwise, the analysis cannot be performed. Precise conventional pointer analysis is often used in program optimizations where conservative assumption must be made - any speculation could result in incorrect execution.

In contrast, our approach does not require the same type of strict correctness. If the behavior of a specific pointer cannot be inferred precisely, we can often speculate or just ignore its effect. For example, if a points-to relation (or location) cannot be inferred statically, we speculatively consider only the other locations gathered in the pointer's location set. We mark the location as undefined. When assigning location sets for the same pointer at a later point in the CFG, one could safely ignore/remove the *undefined location* in the set, if the probability of the pointer accessing that location, at the new program point, is low (e.g., less than 25% in our case).

The main steps of our RB pointer analysis algorithm are as follows: (1) build a control-flow graph (CFG) of the computation, (2) analyze each basic block in the CFG gradually building a PTG, (3) at the beginning of each basic block merge location set information from previous basic blocks, (4) mark locations in the location sets that are unlikely to occur at runtime, at the current program point, as less frequent, (5) mark undefined locations or point-to relations; (6)

repeat steps 2-5 until the PTG graph does not change (i.e., full convergence is reached) or until the allowed number of iterations are reached.

Library calls that may modify pointer values and for which source codes are not available are currently speculatively ignored. If a pointer is passed in as an argument, its location set after the call-point in the caller procedure will be marked as speculative, signaling that the location set of the pointer might be incomplete after the call. In none of the programs we have analyzed we have found library modified pointer behavior to be a considerable factor in gathering precise pointer reuse information.

RB Distance Analysis gathers stride information for pointers changing across loop iterations. This stride information is used to predict pointer-based memory access patterns, and speculation is performed whenever the stride is not fixed. As strides could change in function of the paths taken in the Control-Flow Graph (CFG) of the loop body, only the most likely strides (based on static branch prediction) are considered.

In the example shown in Fig. 1(a), the value of pointer p changes after each iteration. In general, there are two ways to deal with this situation if implemented as part of pointer analysis. Each element in the array structure could be treated as a different location, or, another approach would be to treat the whole array *arr* as a single location. The former is too complicated for compiler analysis while the latter is not precise enough.

In our approach, as shown in Fig. 1(a), we first find the initial location for p. Then, when we find out that p is changing for each iteration, we calculate the distance (stride) between the current location and the location after modification. If the distance is constant, we will use both the initial location and distance to describe the behavior of the pointer.

Extracting stride information is not always easy. In Fig. 1(a), we can easily calculate that the stride for pointer p is 4 bytes. However, for the example in Fig. 1(b), the stride for pointer p is variable since we do not know what value procedure *foo()* will return. In this case, we can use speculation based on static

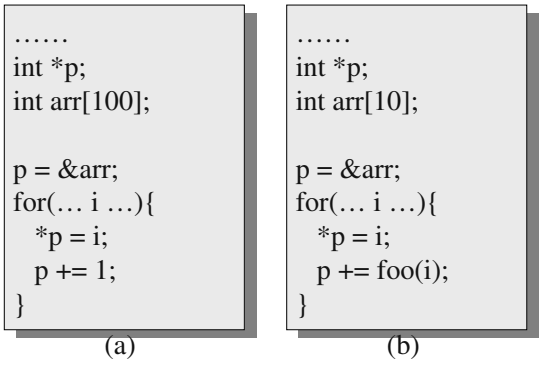


Fig. 1. Distance analysis examples: (a) static stride (b) variable stride

information related to the location set to estimate the stride. For example, the information we do know is (1) p points to array *arr* and (2) the size of array *arr* is small. Based on this information, we can speculate that the stride of p is small although we do not know the exact number.

Another example of stride prediction, as also mentioned earlier, is ignoring strides that are less likely to occur at runtime based on static branch prediction. Clearly, depending on which path is executed at runtime the stride of a pointer might change across loop iterations, as not all the possible paths leading to that pointer access are equally likely to occur.

RB Reuse Analysis attempts to discover those pointer accesses that have reuse, i.e., refer to the same cache line. Reuse analysis uses the information provided by the previous analyses to decide whether two pointer accesses refer to the same cache line. Based on the reuse patterns, pointer accesses are partitioned into reuse equivalence classes. Pointers in each equivalence class have a high probability of referring to the same cache line during execution and will be mapped through the static access path in the Cool-Mem system.

Reuse analysis for array-based accesses has been studied and used in [9, 15, 16]. For pointer-intensive programs, we use a classification scheme similar to theirs, but we redefine it specifically in the context of pointer-based accesses.

1. *Temporal Reuse*: This is the case when a pointer is not changing during loop iterations. This is the simplest case for loop-based accesses.
2. *Self-Spatial Reuse*: If a pointer is changing using a constant stride and the stride is small enough, two or more consecutive accesses will refer to the same cache line.
3. *Group-Spatial Reuse*: A group of pointers can share the same cache line during each loop iteration even when they do not exhibit self-spatial reuse.
4. *Simple-Spatial Reuse*: This exists between two pointers that refer to the same cache line but do not belong to any loop. Simple-spatial reuse is added as a new reuse category because we find that this situation is important for pointer-based programs although it is not as important for array-based programs. The reason for this is that array structures are typically accessed using loops, while pointer-based data structures are often accessed using recursive functions.

Pointer-based memory accesses are partitioned into different *reuse equivalence classes* based on the reuse classification and strides. A reuse equivalence relation exists between two memory accesses if one of the above mentioned reuse relations exists between them. Intuitively, each reuse equivalence class contains those pointer accesses that have a good chance to access the same cache line.

Once we have the reuse equivalence classes, we use a *reuse probability threshold* to decide which of the equivalence classes will likely have high cache line reuse. All the accesses assigned to an equivalence class with a reuse probability smaller than this threshold or not assigned to a class, will be regarded as irregular. In our experiments, we choose the reuse threshold such that the statically estimated reuse misprediction rate is predicted to be smaller than 33%

(the overall misprediction rate could be much lower depending on the mixture of equivalence classes, but could also be larger due to the speculative nature of the information this analysis is based on).

After RB reuse analysis, all the accesses which fall into one of the four reuse categories are regarded as having good reuse possibility. Pointer accesses which have bad locality and small reuse chances are identified as irregular accesses.

3 Application: Compiler-Managed Memory Systems

The results of run-time biased reuse analysis can be applied to general-purpose compiler-enabled cache management systems. In this paper, we replicated a compiler-enabled energy efficient cache management framework, Cool-Mem [2], and extended it by incorporating our pointer reuse analysis techniques. We will give a simple introduction of Cool-Mem architecture in this section, detail information about Cool-Mem can be found in [2].

3.1 Cool-Mem Memory System

Figure 2 presents an overview of the Cool-Mem memory system, with integrated static and dynamic access paths. Cool-Mem extends the conventional associative cache lookup mechanism with simpler, direct addressing modes, in a virtually tagged and indexed cache organization. This direct addressing mechanism eliminates the associative tag-checks and data-array accesses. The compiler-managed speculative direct addressing mechanisms uses the hotline registers. Static mispredictions are directed to the CAM based Tag-Cache, a structure storing cache line addresses for the most recently accessed cache lines. Tag-Cache hits also directly address the cache, and the conventional associative lookup mechanism is used only on Tag-Cache misses.

The conventional associative lookup approach requires 4 parallel tag-checks and data-array accesses(in a 4-way cache). Depending on the matching tag, one of the 4 cache lines is selected and the rest discarded. Now for sequences of accesses mapping to the same cache line, the conventional mechanism is highly redundant: the same cache line and tag match on each access. Cool-Mem reduces this redundancy by identifying at compile-time, access likely to lie in the same cache line, and mapping them speculatively through one of the hotline registers (step 1 in Fig. 2).

Different hotline compiler techniques are used to predict which cache accesses are put into which hotline registers. A simple run-time comparison (step 2) reveals if the static prediction is correct. The cache is directly accessed on correct prediction (step 3), and the hotline register updated with the new information on mis-predictions.

Another energy-efficient cache access path in Cool-Mem is the CAM-based Tag-Cache. It is used both for static mis-prediction (hotline misses) and accesses not mapped through the hotline registers, i.e. dynamic accesses (step 4). Hence it serves the dual-role of complementing the compiler-mapped static accesses

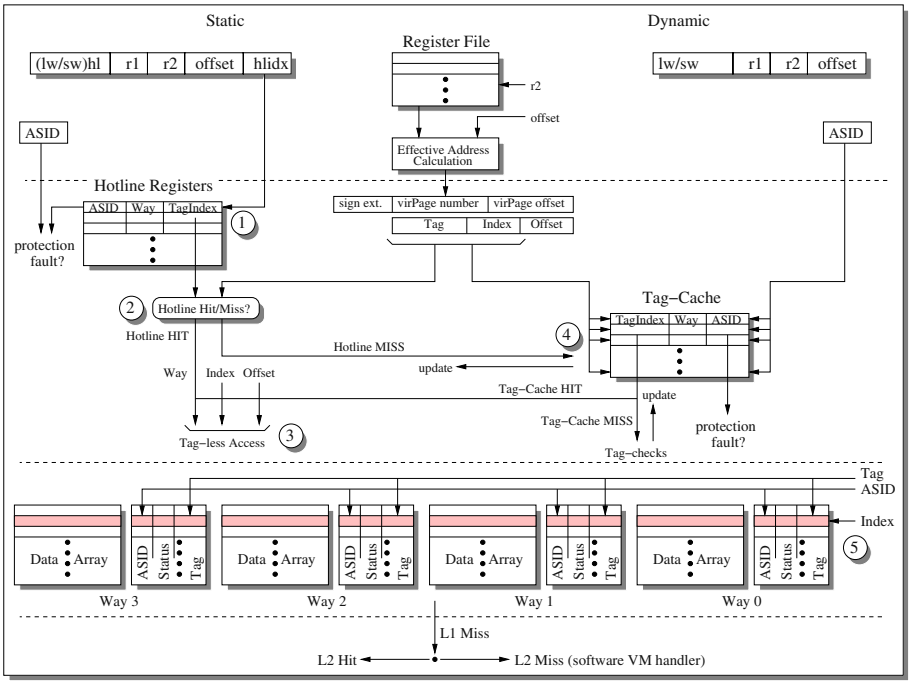


Fig. 2. Cool-Mem Architecture

by storing cache-line addresses recently replaced from the hotline registers, and also saving cache energy for dynamic accesses; the cache is directly accessed on Tag-Cache hits(step 3).

Although the Tag-Cache access is very quick, we assume that the Tag-Cache, accessed on hotline misses, require another cycle, with an overall latency similar to a regular cache access. A miss in the Tag-Cache implies that we fall back to the conventional associative lookup mechanism with and additional cycle performance overhead (step 5). The Tag-Cache is also updated with new information on misses. As seen in Fig. 2, each Tag-Cache entry is exactly the same as a hotline register, and performs the same functions, but dynamically.

3.2 Cool-Mem Compiler

Cool-Mem compiler is responsible for identifying groups of accesses likely to map to the same cache-line, and mapping them through one of the hotline registers. Hotline passes are implemented in two different compiler techniques: (1) Optimistic Hotlines, where the compiler tries to map all accesses through the hotline registers, and (2) Conservative Hotlines, which maps a subset of the accesses that are more regular in nature and as a result, are likely to cause fewer mis-predictions. The description of both algorithms can be found in [2].

Both the optimistic and conservative hotline approaches are not dealing with pointer variables, because pointer information is unknown without pointer alias analysis or points-to analysis. Runtime biased pointer reuse analysis results can be applied easily in the context of the Cool-Mem architecture. Simply, pointer accesses in reuse equivalence classes with reuse attributes larger than the reuse threshold are mapped to static energy-saving cache access paths. At the same time, irregular pointers identified during reuse analysis will be directed to regular cache access paths to avoid energy and performance penalties.

4 Experimental Framework

The SUIF [13]/Machsuif [11] suite is used as our compiler infrastructure. RB pointer and distance analysis is implemented as a SUIF pass which analyzes an intermediate SUIF file and then writes the pointer and stride information back as annotations. RB reuse analysis runs after the pointer analysis pass and writes reuse equivalence class information to the SUIF intermediate file.

The source files are first compiled into SUIF code and merged into one file. All high-level compiler analysis passes, including the pointer and reuse analysis passes, operate at this stage. The annotations from SUIF files are propagated to an *Alpha* binary file through the intermediate stages. We use the SimpleScalar [5] simulator with Wattch [4] extensions for collecting performance and energy numbers. This simulator, capable of running statically linked alpha binaries, has been modified to accommodate the Cool-Mem architecture.

We assume a 4-way in-order Alpha ISA compatible processor and 64 Kbyte 4-way set-associative L1 caches, 0.18 micron technology, and 2.0V V_{dd}. We account for all the introduced overheads and static mispredictions in the architecture as described in [2].

We simulated a number of benchmarks during the selection process, including SPEC 2000 [1], Olden pointer-intensive benchmark suite [10] and several benchmarks used previously by the pointer analysis community [3, 6]. We chose seven benchmarks (shown in Table 1) which contain at least 25% of pointer accesses at runtime.

Table 1. Benchmarks used in simulation results

Benchmark	Source	Description
backprop	Austin	Neural network training
em3d	Olden	Elect. magn. wave propag.
ft	Austin	Minimum spanning tree
ks	Austin	Graph Partition
08.main	McGill	Polygon rotation
mcf	SPEC2000	Combinatorial optimization
09.vor	McGill	Voronoi diagrams

5 Results

In this section, we show experimental results for the above benchmarks, including benchmark statistics as well as energy saving results collected using Wattch.

5.1 Regular Versus Irregular Pointers

Identifying those pointers which do not have good locality is important because they normally result in energy and performance penalties when managed statically. Figure 3(a) shows the percentage of irregular pointers found during static compiler analysis. Different programs have a different portion of irregular pointers. In some of them, such as *main* and *em3d*, up to 99% of all the pointers are predicted as regular. Other programs like *ft* have almost 80% irregular accesses.

Figure 3(b) shows the misprediction rate of the pointers predicted when mapped to the static cache access path. The misprediction rate refers to the accesses that do not point to the cache line predicted. As shwon in the second bar, the misprediction rate for irregular pointers, if mapped through the energy efficient cache access path, is very high for most of the programs. It is at least twice the misprediction rate of those pointers we identified as regular pointers. The only exception is *backprop*, which operates on a relatively small data structure, such that all the pointer accesses have very good locality. However, we can see that the misprediction rate for irregular pointers in *backprop* is still much higher than those of regular pointers.

We also show the misprediction rate for the case when all the pointer accesses are mapped through the static path. Note that the misprediction rate is significantly reduced by removing the irregular pointer accesses. For *em3d*, the overall misprediction rate is reduced by almost 50% while identifying only 1.7% of all

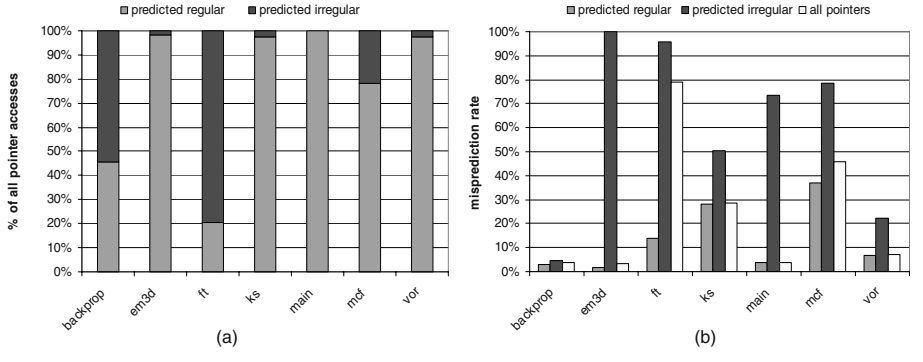


Fig. 3. Regular versus irregular pointers: (a) runtime percentage of statically determined regular and irregular pointers; (b) Static misprediction rates of pointer accesses when mapped to Cool-Mem's static cache access path. Misprediction occurs when a pointer that is predicted to have high reuse statically will not access the predicted cache line at runtime

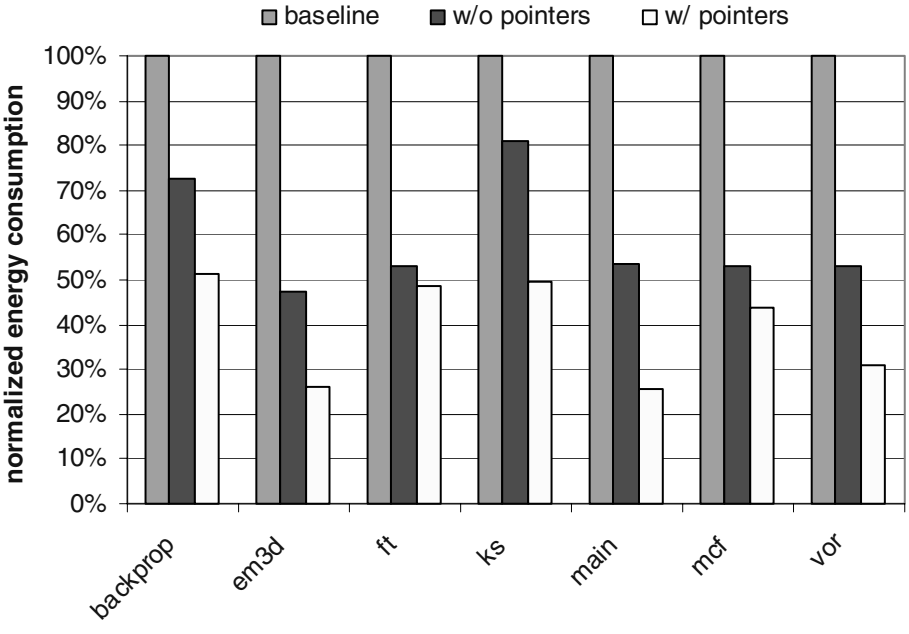


Fig. 4. Normalized L1 D-cache energy consumption

the pointer accesses as irregular. We also identified almost 80% of all pointers as irregular in *ft*, which have a misprediction rate greater than 95%.

5.2 Energy Savings

Figure 4 shows the energy consumption results which are normalized to the unoptimized hardware baseline architecture. The baseline energy number, which is shown as 100%, is the first bar. The second bar shows the normalized energy consumption by applying the published Cool-Mem techniques without mapping the pointer-based accesses through the statically managed cache access path. Finally, the energy consumption number, which uses the results of our reuse analysis for pointer-based accesses, is shown last. Compared to the optimization which maps only array-based accesses, 4% to 31% extra energy reduction is achieved on the L1 data cache energy consumption by mapping the pointer-based memory accesses that are statically predicted as regular through the statically managed cache access path.

6 Conclusion

Compiler-enabled cache management for pointer-intensive programs is challenging because pointer analysis is difficult and sometimes even impossible for large or complex programs. By applying the runtime biased pointer analysis techniques, we can always complete analysis for any pointer-intensive program with-

out any constraints. The techniques proposed increase the fraction of memory accesses that can be mapped statically to energy efficient cache access paths and shows significant additional energy reduction in the L1 data cache.

Our future work includes further investigation and experiments on the runtime biased pointer analysis approach and applying the analysis to other compiler-enabled techniques such as compiler-directed prefetching on pointer-intensive codes.

References

1. The standard performance evaluation corporation, 2000. http://www.spec.org.
2. R. Ashok, S. Chheda, and C. A. Moritz. Cool-mem: Combining statically speculative memory accessing with selective address translation for energy efficiency. In *ASPLOS*, 2002.
3. T. Austin. Pointer-intensive benchmark suite, version 1.1, 1995. http://www.cs.wisc.edu/~austin/ptr-dist.html.
4. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, Vancouver, British Columbia, June 12–14, 2000. IEEE Computer Society and ACM SIGARCH.
5. D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
6. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
7. M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 726–731, Los Alamitos, CA, June 15–19 1998. ACM/IEEE.
8. J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stephany, and S. C. Thierauf. A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor. *Digital Technical Journal of Digital Equipment Corporation*, 9(1), 1997.
9. T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching*. PhD thesis, Dept. of Computer Science, Stanford University, Mar. 1994.
10. A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.
11. M. Smith. Extending suif for machine-dependent optimizations. In *Proc. First SUIF Compiler Workshop*, Jan. 1996.
12. O. S. Unsal, R. Ashok, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-cache for hot multimedia. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–283. IEEE Computer Society, 2001.
13. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. Lam, and J. L. Hennessy. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

14. E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 124–133, Austin, Texas, Dec. 1–5, 2001. IEEE Computer Society TC-MICRO and ACM SIGMICRO.
15. M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, Aug. 1992.
16. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

Inter-program Compilation for Disk Energy Reduction*

Jerry Hom and Ulrich Kremer

Rutgers University, Department of Computer Science,
Piscataway, NJ 08854, USA
{jhom, uli}@cs.rutgers.edu

Abstract. Compiler support for power and energy management has been shown to be effective in reducing overall power dissipation and energy consumption of individual programs, for instance through compiler-directed resource hibernation and dynamic frequency and voltage scaling (DVS). Typically, optimizing compilers perform intra-program analyses and optimizations, i.e., optimize the input program without the knowledge of other programs that may be running at the same time on the particular target machine. In this paper, we investigate the opportunities of compiling sets of programs together as a group with the goal of reducing overall disk energy. A preliminary study and simulation results for this inter-program compilation approach shows that significant disk energy can be saved (between 5% and 16%) over the individually, disk energy optimized programs for three benchmark applications.

1 Introduction

Handheld computers have come a long way from just being a sophisticated address book and calendar tool. Handheld computers or pocket PCs feature rather powerful processors (e.g. 400MHz XScale), 64MB or more of memory, wireless Ethernet connections, and devices such as cameras, speakers, and microphones. They are able to run versions of standard operating systems such as Linux and Windows. However, as compared to their desktop PC counterparts, pocket PCs have significantly less resources, in particular power resources, and less computational capabilities. While pocket PCs will evolve further in terms of their resources and capabilities, they will always be more resource constrained than the comparable desktop PCs, which will evolve as well. As a result, users of pocket PCs have to be more selective in terms of the programs that they wish to store and execute on their handheld computer. Such a program set may include a web browser, an mpeg player, communication software (e.g. ftp), a voice recognition system, a text editor, an email tool, etc. Typically, a user may only run a few of these programs at any given point in time. Figure 1 shows an example of possible subsets of programs executing simultaneously on a handheld PC or

* This work has been partially supported by NSF CAREER award #9985050.

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 13–25, 2004.

PDA. Only program combinations that occur frequently or are considered important are represented as explicit states. For example, running a web browser, an audio player, and an email program at the same time may occur frequently enough to promise a benefit from inter-program analysis and optimizations. State transitions are triggered by program termination and program activation events. The state marked "???" is a catch-all state that allows internal transitions and represents all combinations of simultaneous program executions that are not considered interesting enough to be analyzed and optimized as a group. The graph shows the underlying assumption of our presented work, namely that a typical usage pattern of a handheld PC or PDA can be characterized by a limited number of program subsets where the programs in a subset are executed together. This makes optimizing particular states or program combinations feasible. The graph can be determined through program traces or other means. Techniques and strategies to determine such a graph are beyond the scope of this paper.

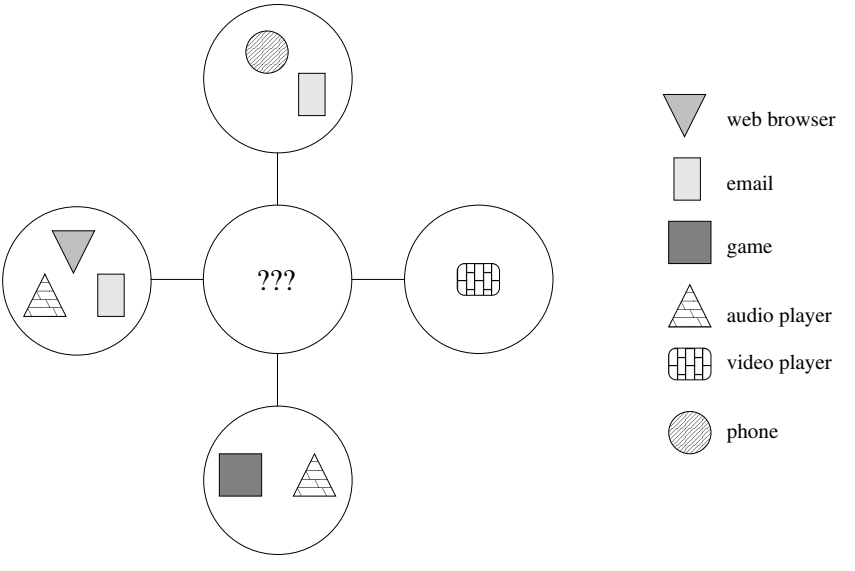


Fig. 1. Example finite state machine. Nodes represent subsets of applications executing simultaneously. Transitions are triggered by program activation and termination events. The catch-all state marked "???" allows internal transitions and represents all non-interesting program combinations

A compiler is able to reshape a program's execution behavior to efficiently utilize the available resources on a target system. Traditionally, efficiency has been measured in terms of performance, but power dissipation and energy consumption have become optimization goals in their own right, possibly trading-off power and energy savings for performance. One effective technique to save power and energy is resource hibernation which exploits the ability of devices to switch between different activity states, ranging from a high activity (active) to a deep

sleep state. As a rule of thumb, the lower the activity state, the more power and energy may be saved, but the longer it will take to bring back the device into the fully operational, active state. Each transition between an activity state has a penalty, both in terms of performance and power/energy.

An energy-aware compiler optimization can reshape a program such that the idle times between successive resource accesses are maximized, giving opportunities to hibernate a device more often, and/or in deeper hibernation states. This compilation strategy has been shown to work well in a single process environment[1, 2], but may lead to poor overall results in a multiprogramming environment. In a multiprogramming setting, one program may finish accessing a resource and direct the resource to hibernate during some time of idleness. During this time, a second program needs to access the resource. In the worst case, each program alternately accesses a resource such that the resource never experiences significant amounts of idleness. In effect, one program's activity pattern interferes with another program's idle periods and vice versa. To alleviate this problem, some inter-program or inter-process coordination is necessary.

Operating systems techniques such as batch scheduling coordinate accesses to resources across active processes. Requests for a resource are grouped and served together instead of individually, potentially delaying individual requests for the sake of improved overall resource usage. In contrast to operating systems, compilers have often the advantage of knowing about future program behavior and resource requirements. Instead of reacting to resource requests at runtime, a compiler can insert code into a set of programs that will proactively initiate resource usage across the program set at execution time. This is typically beyond the ability of an operating system since it requires program modifications and knowledge about future resource usage.

In this paper, we investigate the potential benefits of an inter-program optimization strategy for disk power and energy management. This paper focuses on a compiler/runtime library based approach, although an OS only, or a combined OS and compiler approach is also possible. An initial study of a compiler only vs. OS only strategy for inter-program optimizations is currently underway.

By considering multiple programs, the compiler applies a synchronization optimization which we call *inverse-barrier*. Previous work has shown that applications which read data from disk in a streamed fashion (i.e., periodic access) can utilize large disk buffers to save energy[1]. These disk buffers are local to each application and serve to increase the idle period between disk accesses. Hence each application has a unique disk access interval associated with the size of its buffer. Having longer intervals between disk accesses creates opportunities to hibernate the disk. This intra-program optimization works well for applications running in isolation, but when multiple such applications execute simultaneously, some of the intra-program optimization's effects are negated. That is, the disk idle period of one application is interrupted by a disk access from another application. This will occur whenever the intervals between accesses by multiple applications are different.

Simulation experiments using physical traces of three intra-program optimized applications show significant energy savings when applying the inverse-barrier optimization. The inverse-barrier also proves more effective at saving energy and maintaining performance than using barrier synchronization.

2 Related Work

This research is related to a few OS level techniques. In order for inter-program compilation to be useful, it should apply optimizations which span across applications. The initial idea began from the notion of a programming mechanism called a barrier to delay disk accesses in order to cluster them as well as increase the idle time between accesses. To be useful, the OS must support such a programming paradigm with a scheduling policy. Indeed, co-scheduling is one example and a well-known technique for scheduling processes in a distributed group at the same time[3]. One aspect is to schedule associated processes at the same time thereby letting processes make progress within their scheduled timeslot. Since our work relies on idleness to save energy, we desire processes to synchronize by scheduling their resource accesses together and maximizing idleness.

Our mechanism for synchronizing accesses, inverse-barrier, is similar to implicit coscheduling for distributed systems[4]. Dusseau et al. introduce a method for coordinating process scheduling by deducing the state of remote processes via normal inter-process communication. The state of a remote process helps the local node determine which process to schedule next. Inverse-barrier applies this idea to coordinate resource accesses by multiple processes on a single system.

More recently, Weissel et al. developed Coop-I/O to address energy reduction by the disk[5]. Coop-I/O enables disk operations to be deferrable and abortable. By deferring operations, the OS may batch schedule them at a later time until necessary. The research also shows some operations may be unnecessary and hence the abortable designation. However, the proposed operations require applications to be updated by using the new I/O function calls. In contrast, our technique utilizes compiler analysis to determine which operations should be replaced. The modification cost is consolidated to the compiler optimization and a recompile of the application.

In terms of scheduling paradigms, this work resembles basic ideas from the slotted ALOHA system[6, 7]. The essential idea is to schedule access between multiple users to a common resource (e.g. radio frequency band) while eliminating collisions or when multiple host transmit on the same frequency at the same time. For our purposes, a collision takes on almost the opposite notion of a disk request without any other requests close in time. Rather than scheduling for average utilization of the disk, optimizing for energy means scheduling for bursts of activity followed by long periods of idleness.

A form of inter-program compilation has been applied to a specific problem of enhancing I/O-intensive workloads[8]. Kadayif et al. use program analysis to determine access patterns across applications. Knowledge of access patterns

allows the compiler to optimize the codes by transforming naive disk I/O into collective or parallel I/O as appropriate. The benefit manifests as enhanced I/O performance for large, parallel applications. We aim to construct a general framework suitable for developing resource optimizations across applications to reduce energy and power consumption.

3 Intra- Versus Inter-program Optimizations

Handheld computing devices may be designed as general purpose, yet each user may desire to run only a certain mix of applications. If this unique set of applications remains generally unchanging, compiling the set of applications together with inter-program scheduling can enhance performance and cooperation by synchronizing resource usage. A further goal is to show how new energy optimizations may be applied for resource management.

Consider a scheduling paradigm across programs on a single processor. The proposed optimizations augment the paradigm with user-transparent barrier and *inverse*-barrier mechanisms to resemble thread scheduling. Barrier semantics enforce the notion that processes or threads (within a defined group) must pause execution at a defined barrier point until all members of the group have reached the barrier. The notion of an inverse-barrier applies specifically to resources. That is, when a process or thread reaches an inverse-barrier (e.g. by accessing a resource), all members of the group are notified to also access the resource. Synchronizing resource accesses eliminates any pattern of random access and allows longer idle periods where the resource may be placed in a low power hibernation mode.

An example of an intra-program optimization is a transformation to create large disk buffers in memory thereby increasing the disk's idle time for hibernation. While application transformations have been shown to benefit applications executed in isolation[1], running such locally optimized programs concurrently squander many of the benefits because the access pattern from each process disrupts the idle time of the resource. This intra-program optimization considers each program by itself while an inter-program optimization now considers all programs in a group and augments them to cooperate in synchronizing accesses to a resource.

Program cooperation can be accomplished in at least two ways: (1) delay resource access until all group members wish to use it or (2) inform all group members to use the resource immediately. The first method is similar to a barrier mechanism in parallel programming and can be used by programs which lack deadlines. The second method has the notion of an inverse-barrier and can be used by programs with deadline constraints such as real-time software.

Programs using a barrier cooperate in a passive fashion. When a program wants to access a resource, it will pause and wait until all members in its group also wish to access the resource. When all members reach the barrier, then they all may access the resource consecutively. To avoid starvation, each waiting

process has a timer. If the timer expires, the process will proceed to access the resource.

Programs using an inverse-barrier cooperate actively to synchronize resource accesses. When a program needs to access a resource, it will notify all members in its group to also access the resource immediately. This has the effect of refilling a program's disk buffer earlier than necessary which ensures that deadlines are satisfied. This synchronization mechanism can be communicated via signals among all processes. Only those processes with an appropriately included signal handler will follow suit in accessing the resource. Uninterested processes may simply ignore the signal.

The signal mechanism is also used in our current compilation framework to inform active programs about other active, simultaneously executing programs. The compiler generates signal handling code within each program that implements state transitions between interesting groups of applications. Each time a program is about to terminate, it sends a signal to inform other active processes about its termination event. The appropriate signal handlers in the remaining active programs will then make the corresponding state transition. Each time a program begins execution, it sends a signal to inform other active programs about its presence. In return, active programs will send a signal informing the "new" process about the state that they are in, i.e., inform the program about its current execution group. This way, a program is aware of its group execution context, and can perform appropriate optimizations in response to inverse-barrier signals.

4 Experiments and Results

This benefit analysis builds upon previous work and examines three streaming applications *mpeg_play*, *mpg123*, and *sftp*. The MPEG video and audio decoders are examples of real-time applications where they must have low latency access to the disk. They cannot afford to wait for other applications before accessing the disk. On the other hand, ftp is a silent process, mostly invisible to the user, and can tolerate pauses with the understanding that throughput performance is traded off with energy savings.

From these three applications, there are three experiments with interesting results: 1) all three applications, 2) video with audio, 3) audio with ftp. Combining video with ftp is expected to produce similar results as (3). Although the original experiments operated on the same file, each run produced slightly different traces because of the dynamic nature of the disk profiling at program startup. However, the variance from each set of runs was minor and demonstrates the stability of the profiling strategy. All experiments used disk traces from Heath et al.[1] for hand simulating the behavior of these programs executing at the same time with and without inter-program optimizations applied.

The disk traces were modified to better simulate the more interesting, steady state conditions while the applications are running simultaneously. The duration of the traces from the three programs vary considerably. For example, the

trace for *mpg123* lasted 425 seconds while *mpeg_play* and *sftp* were 106 and 232 seconds, respectively. The shorter traces were extended to be roughly time equivalent to *mpg123*. Since each program was optimized to produce periodic disk accesses, extending the execution time is merely a matter of using larger data files. Hence, the traces were extended by "copying and pasting" multiples of disk access periods.

A second modification deals with the buffer sizes. The buffer size is calculated at runtime after some profiling steps. A prudent calculation would divide the buffer size by n where n is the number of applications compiled with this disk buffer optimization. Otherwise, if all applications used its maximums buffer size, thrashing may occur when such applications are actually executed together. Thus, the disk access intervals for each application was divided by either 2 or 3 for the experiments.

These particular applications lightly stress the CPU, and the experiments assume that the CPU meets all deadlines (e.g. decoding frames) for all applications running simultaneously. The CPU can decode all frames of video and audio while copying blocks of data for file transfer without degrading performance. Degraded performance might result in dropped frames. However, the physical disk is constrained to serving one process at a time. Thus, if more than one process issues a disk request at the same time, they will be queued and interleaved. In effect, disk access time by processes cannot be overlapped and hidden.

The results of these experiments are closely tied to system parameters. A different disk will change the mix of thresholds in determining when to switch power states, but the essential premise is the potential to save energy by utilizing low power states. Table 1 summarizes the parameters measured on a real disk[1]. When transitioning from *idle* to *standby*, the disk spends 5.0 seconds in the *transition* state. When waking up from *standby* to *idle* or *read*, the disk takes 1.6 seconds. The idleness threshold at which transitioning to *standby* becomes profitable is 10.0 seconds. That is, when the system knows the next disk access is greater than 10.0 seconds, the system should tell the disk to transition to *standby*.

Table 1. Disk states and power levels

Disk States	Power (W)
wakeup	3.0
read	1.8
idle	0.9
transition	0.7
standby	0.2

The first experiment combines all three applications. The respective disk access periods are $P_{mpeg\_play} = 11.7$, $P_{mpg123} = 13.7$, and $P_{sftp} = 23.5$; all times in seconds. Figure 2 shows the disk access traces of each application. The top

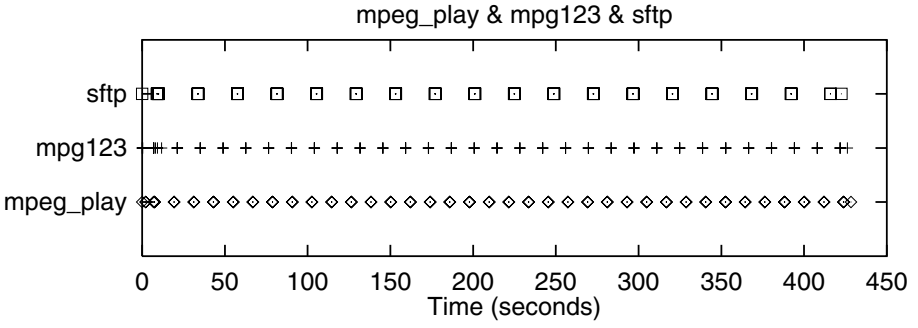


Fig. 2. Disk access traces for *mpeg_play*, *mpg123*, and *sftp*

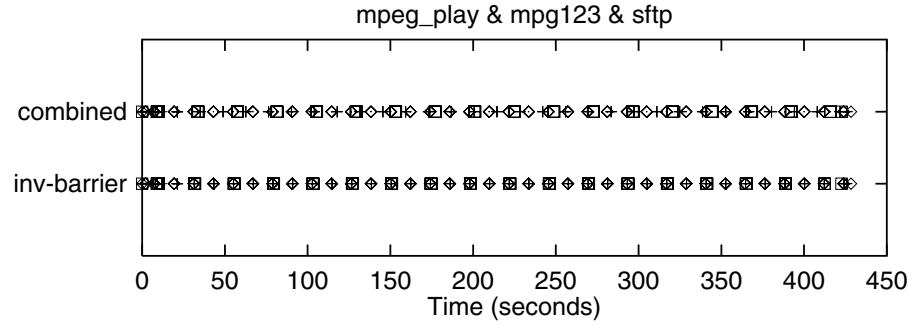


Fig. 3. Running all three applications simultaneously. Comparison of disk access patterns with inverse-barrier optimization for synchronization. Inverse-barrier scheduling saves 5.4% energy

row of Figure 3 shows an overlay of all disk accesses. The bottom row shows the synchronization when using inverse-barrier scheduling. Net energy savings is 5.4%.

The second experiment combines *mpeg_play* with *mpg123*. The disk access periods are $P_{mpeg\_play} = 17.6$ and $P_{mpg123} = 20.6$. Figure 4 illustrates the idleness interference patterns of just two applications vs. inverse-barrier synchronization. A similar pattern can be seen with the third experiment in Figure 5. Here, the disk access periods are $P_{mpg123} = 20.6$ and $P_{sftp} = 35.2$. The energy saved in these two experiments are 15.9% and 9.8%, respectively.

There is a key difference between the inverse-barrier and barrier mechanisms. An OS may employ a barrier mechanism to delay resource accesses for applications which can tolerate such latencies. For example, *sftp* has few constraints about deadlines since it operates by best effort semantics over an unreliable network. If an OS uses this assumption to schedule *sftp* with barriers at disk accesses (i.e., delaying until next access by another application), there will certainly be a performance delay. This mode of operation can still save energy by batching the disk access but also depending on how much delay is involved. The difference with inverse-barrier is the pre-emptive action to ensure that buffers are always

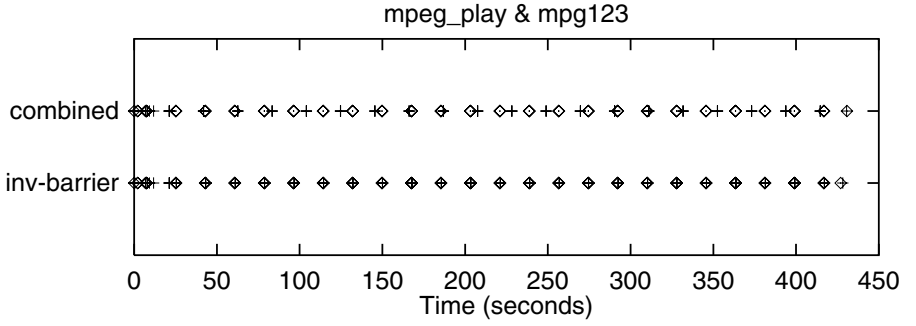


Fig. 4. Comparison of two applications' (*mpeg_play*, *mpg123*) disk access patterns. Inverse-barrier scheduling saves 15.9% energy

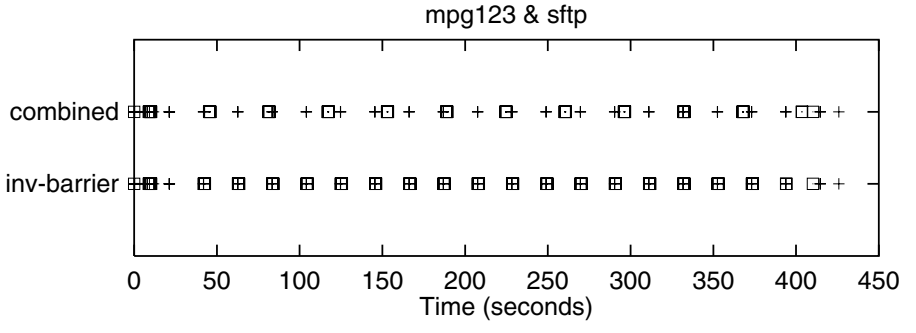


Fig. 5. Comparison of two applications' (*mpg123*, *sftp*) disk access patterns. Inverse-barrier scheduling saves 9.8% energy

sufficiently full. At every synchronized disk access, each process can check the data capacity of its buffer and decide whether to read more data. Another optimization during the buffer check might compute the differential between resource access periods. For instance, if a process has a resource access period more than twice as long as the current period, it can afford to skip every other resource access and maintain a non-empty buffer.

The last experiment explores the behaviors of barrier scheduling; Figure 6 illustrates the difference. In terms of execution time, *sftp* finishes over two minutes later using the barrier vs. the inverse-barrier. Compared to the baseline of running all applications together under normal scheduling, the barrier method expends 2.4% more energy. Barrier scheduling is only slightly more expensive in energy yet can impact performance. In this case, *sftp*'s performance is delayed by 31.4%. Under inverse-barrier scheduling, there is no performance loss while showing modest energy savings.

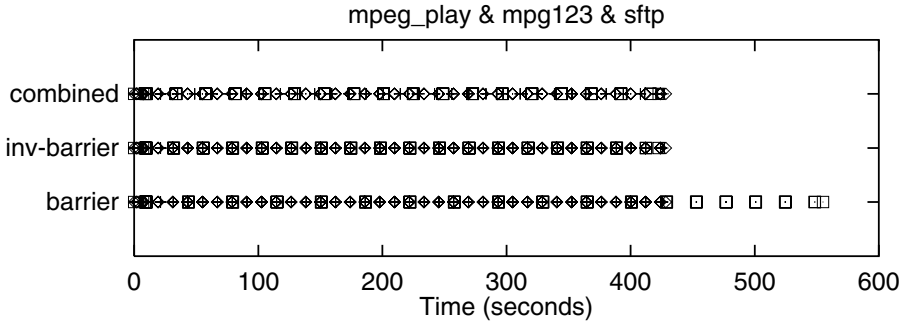


Fig. 6. Comparison of intra-program buffered I/O optimizations, with inter-program inverse-barrier scheduling, and with barrier scheduling on *sftp*. Barrier scheduling causes *sftp* to finish 132 seconds later, a 31.4% performance delay, while using 2.4% more disk energy overall

5 Analysis of Potential Energy Savings

Investigating the upper bounds on energy savings gives an indication whether this avenue of research is worthwhile. Exploring the involved parameters can provide insights into how this technique is beneficial. Towards that end, consider a situation of two programs, A and B, optimized to exhibit periodic resource access behavior. Inter-program compilation with inverse-barrier scheduling results in our optimized execution, but how much can it possibly save over the previous intra-program optimization, which has already shown large disk energy savings? We can approximate this with an analytical examination of the cases where the intra-program execution deviates from the optimal case where all programs access a resource at the same time (i.e., in batch mode). Our inter-program optimization results in such an optimal case. Thus, the difference represents the potential energy savings.

Refer to Table 2 for a list of involved parameters. The following description of these parameters are illustrated in Figure 7. Hence, P_A and P_B represent the period between resource accesses by programs A and B; assume $\frac{P_A}{2} < P_B < P_A$ and let $\Delta P = P_A - P_B$. Each program accesses the resource for an amount of time, R_A and R_B. The rise and fall in the graphs of programs A and B merely indicate a resource access. Only one resource is considered, so its corresponding hibernation threshold time will be designated simply H. If a resource will be idle for at least H, then hibernation will be beneficial and assumed to be initiated. Consequently, we assume $min(P_i) > H$; otherwise any chance for hibernation is gone.

There are three ways to categorize the resource access patterns, demonstrated by the three accesses of A (A^1, A^2, A^3) along with the four accesses of B (B^1, B^2, B^3, B^4). A^1 is *optimal* because it is clustered with B^1. Since the resource will not be used again within H of A^1, hibernation may be initiated immediately. A^2 is *sub-optimal* because it occurs within H of B^2. The accesses are mildly offset, and the resource consumes extra energy by remaining in the idle power

Table 2. Analytical parameters

Variable	Description
P_i	resource access Period of program i
R_i	length of access (Read) time by program i
H_i	Hibernation threshold of resource i
E_i	average Energy use in case i

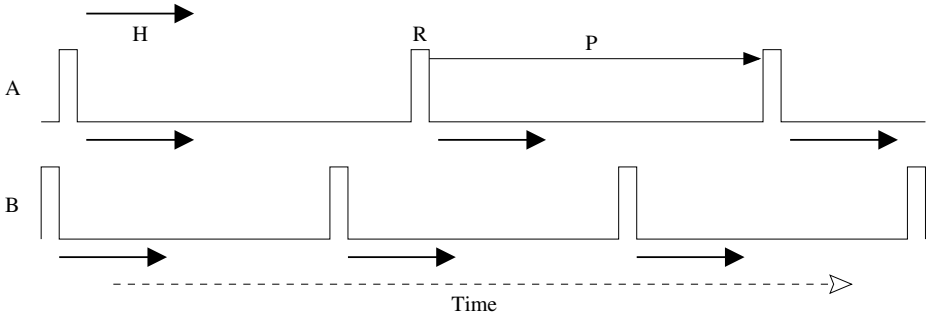


Fig. 7. Resource access patterns of programs A and B

state. A^3 is *out-of-phase* because it occurs after H of B^3, and B^4 occurs after H of A^3. The effect is that B^3's and A^3's hibernation periods are immediately interrupted. There is little opportunity to save energy during the respective hibernation periods. The next question is, what percentage of time do each of the three cases occur?

The optimal case, *opt*, can be expected to occur $\frac{\Delta P}{P_B}\%$ of the time. If A and B have a synchronized access, then each respective access afterward will be offset by ΔP. They will coincide again after $\frac{P_B}{\Delta P}$ accesses. The sub-optimal case is expected to occur $\textit{sub-opt} = (max(\frac{2H}{P_B}, 1) - \textit{opt})\%$ of the time. The first term refers to all accesses within H of an access, including the optimal case. Subtracting the optimal case gives just the sub-optimal case. Lastly, the out-of-phase case occurs $\textit{out} = (1 - (\textit{opt} + \textit{sub-opt}))\%$ of the time, or simply the remaining percentage of time after subtracting the optimal and sub-optimal cases.

The next step toward estimating potential energy savings is calculating the average energy consumed during the three cases and computing the differences. The energy usage for each case can be obtained from a power consumption profile which is a simple graph showing the amount of time spent in the various power states. In the non-optimal cases, there may be many instances of the graphs corresponding to different timing offsets between accesses. These are averaged to produce one profile graph for each of the sub-optimal and out-of-phase cases. Figure 8 shows what a sample power profile may look like. The average energy usage of each case, E_i, is now a matter of summing the power levels over time.

Finally, with the average energy use of all three cases (E_{opt}, $E_{sub-opt}$, E_{out}), an upper bound on potential energy savings can be computed. Energy savings

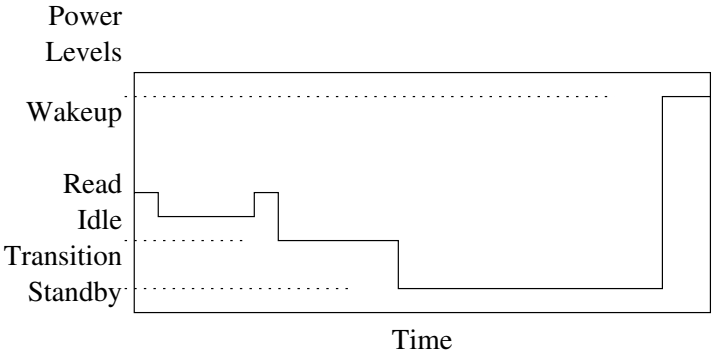


Fig. 8. Sample power profile graph

over the sub-optimal case is $\Delta E_{sub-opt} = E_{sub-opt} - E_{opt}$, and the out-of-phase case is $\Delta E_{out} = E_{out} - E_{opt}$. These cases occur *sub-opt* and *out* percent of the time. Thus, the upper bound is $(\textit{sub-opt} * \frac{\Delta E_{sub-opt}}{E_{opt}}) + (\textit{out} * \frac{\Delta E_{out}}{E_{opt}})$. Applying this analysis to the experiment of running *mpeg_play* and *mpg123* together under inter-program optimization, the upper bound is estimated at 26.2% while actual savings is 15.9%. The upper bound can never be reached because of the small overhead involved during program startup for profiling to initialize the disk buffers. For this experiement, the startup overhead accounted for 5.8% of the total execution time.

6 Summary and Future Work

Inter-program optimization is a promising new compilation strategy for sets of programs that are expected to be executed together. Such sets occur, for instance, in resource restricted environments such as handheld, mobile computers. Resource usage can be coordinated across all programs in the set, allowing additional opportunities for resource hibernation over single program, i.e., intra-program, optimizations alone. This paper discussed the potential benefits of inter-program analysis using the disk as the resource. An analysis of energy savings and simulation results for a set of three benchmark programs show that further significant energy savings over intra-program optimizations (between 5% and 16% for the simulations) can be achieved.

The compiler and OS have unique perspectives on key parts of the entire resource management scheme. We hope to experimentally explore and discover the strengths from each, then apply them in developing a resource-aware compiler and OS system. A current study is trying to assess the advantages and disadvantages of a compiler-only; compiler and runtime system; OS-only; and compiler, runtime system and OS approach to inter-program resource management. We will also be using physical measurements to guide and validate our development efforts.

References

1. Heath, T., Pinheiro, E., Hom, J., Kremer, U., Bianchini, R.: Application transformations for energy and performance-aware device management. In: Proceedings of the Conference on Parallel Architectures and Compilation Techniques. (2002) Best Student Paper Award.
2. Delaluz, V., Kandemir, M., Vijaykrishnan, N., Irwin, M., Sivasubramaniam, A., Kolcu, I.: Compiler-directed array interleaving for reducing energy in multi-bank memories. In: Proceedings of the Conference on VLSI Design. (2002) 288–293
3. Ousterhout, J.: Scheduling techniques for concurrent systems. In: Proceedings of the Conference on Distributed Computing Systems. (1982)
4. Arpaci-Dusseau, A., Culler, D., Mainwaring, A.: Scheduling with implicit information in distributed systems. In: Proceedings of the Conference on Measurement and Modeling of Computer Systems. (1998) 233–243
5. Weissel, A., Beutel, B., Bellosa, F.: Cooperative I/O — a novel I/O semantics for energy-aware applications. In: Proceedings of the Conference on Operating Systems Design and Implementation. (2002)
6. Abramson, N.: The ALOHA system — another alternative for computer communications. In: Proceedings of the Fall Joint Computer Conference. (1970) 281–285
7. Roberts, L.: ALOHA packet system with and without slots and capture. Computer Communications Review **5** (1975) 28–42
8. Kadayif, I., Kandemir, M., Sezer, U.: Collective compilation for I/O-intensive programs. In: Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems. (2001)

Energy Consumption in Mobile Devices: Why Future Systems Need Requirements–Aware Energy Scale-Down

Robert N. Mayo and Parthasarathy Ranganathan

Hewlett Packard Labs,
1501 Page Mill road MS 1177,
Palo Alto California 94304
{Bob.Mayo,Partha.Ranganathan}@hp.com

Abstract. The current proliferation of mobile devices has resulted in a large diversity of designs, each optimized for a specific application, form-factor, battery life, and functionality (e.g., cell phone, pager, MP3 player, PDA, laptop). Recent trends, motivated by user preferences towards carrying less, have focused on integrating these different applications in a single general-purpose device, often resulting in much higher energy consumption and consequently much reduced battery life. This paper argues that in order to achieve longer battery life, such systems should be designed to include *requirements-aware energy scale-down techniques*. Such techniques would allow a general-purpose device to use hardware mechanisms and software policies to adapt energy use to the user's requirements for the task at hand, potentially approaching the low energy use of a special-purpose device. We make two main contributions. We first provide a model for energy scale-down. We argue that one approach to design scale-down is to use special-purpose devices as examples of power-efficient design points, and structure adaptivity using insights from these design points. To understand the magnitude of the potential benefits, we present an energy comparison of a wide spectrum of mobile devices (to the best of our knowledge, the first study to do so). A comparison of these devices with general-purpose systems helps us identify scale-down opportunities. Based on these insights, we propose and evaluate three specific requirements-aware energy scale-down optimizations, in the context of the display, wireless, and CPU components of the system. Our optimizations reduce the energy consumption of their targeted subsystems by factors of 2 to 10 demonstrating the importance of energy scale-down in future designs.

1 Introduction

Recent advances in computing and communication have led to increased use of a large number of mobile computing devices. These devices have many purposes and form-factors including both general-purpose devices like laptops, pocket PCs, and palm computers as well as specialized devices like portable MP3 players and e-mail pagers. Both form factor and energy are critical resources for all these devices, forcing users to trade away functionality to gain smaller form factors and longer battery

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 26–40, 2004.

lifetimes. A variety of such tradeoffs exist for many specific tasks. For instance, the email task can be accomplished with a feature-rich and high-energy application like Microsoft Outlook running on a laptop, or a reduced-feature low-energy application like a Blackberry email pager[1].

The drive for small form factors is strong, resulting in users demanding the most value from a given form factor. The popularity of camera/cell phone combinations are examples of multiple applications in a single device, with a consequent reduction in overall form factor when compared to the sum of two separate devices. Most successfully-converged products combine applications with similar hardware and software requirements. In our example, both cameras and cell phones can share a processor with limited computational power, a common display, and a small set of input buttons. But different applications inherently have some mismatch. Higher-resolution color displays are needed on camera/phone combinations compared to stand-alone mobile phones, leading to increased energy use even when only using the phone portion of the device.

Perhaps the best examples of this mismatch are in general-purpose devices like PDAs. These devices can run a wide range of software and accept a wide range of hardware cards. Thus, they would appear to be the ultimate device for handheld convergence. But this generality currently comes at the cost of high energy use, making the devices much less attractive. The main reason for this is the lack of adaptability in the hardware and software energy use.

Consider a comparison of two email applications: feature-rich email software running on a high-end handheld computer versus an email pager. Both handle the email task well, but with different tradeoffs between functionality and battery life. In the case of the pager, users want long battery life, notification of incoming email, and an acceptable screen for text. Much less important is the ability to view color photographs or read attached files. If the user of a handheld computer desires the reduced features and longer battery life, it is largely unobtainable. It simply is not possible to run a pager-like application on a high end handheld computer and get pager-like battery life.

Application-specific devices like cell phones, email pagers, and MP3 players are examples of highly successful tradeoffs of functionality, form-factor, and battery life. Their success proves them to be excellent points in the design space, in that users find high value in them. As such, they can serve as benchmarks against which we can compare general-purpose devices. A truly general-purpose device should strive to emulate these design points, including not only the specific functionality but also the battery life.

While there are many technical challenges, *requirements-aware energy scale-down* is an approach that can yield improvements. The idea is to have both hardware and software that can scale their features and energy use to meet a variety of design points. For high functionality, hardware and software would present a rich set of features to

[1] In this paper, we use the terminology *task* to mean a broad category, such a music listening task or an email task. A specific set of hardware and software to accomplish this task is called an *application*, and involves various tradeoffs of functionality, battery life, and form factor. For instance, the music listening task could be accomplished using either a PDA or a tiny MP3 player.

the user, at the cost of higher energy use. At the low end, hardware and software would scale their functionality to match a popular low-end design point, resulting in lower energy use. Specifically, we recommend considering each component in the general-purpose device and comparing it to the requirements of the applications using that device. Ideally, each general-purpose component should be capable of scaling down its energy use to match the design point used by the application with the lowest requirements. There are two alternatives for achieving this: *gradation-based scaling*, where the component itself has a wide range of adaptability, or *plurality-based scaling*, where the device chooses among multiple components with different properties. We suggest that a good way of determining mismatches in the component's functionality and the task requirements for a general-purpose device is to use as reference the component's functionality in a special-purpose device targeted at that application.

The rest of the paper is organized as follows. The next section of the paper (Section 2) further discusses the need for and potential from energy scale-down. In particular, it characterizes the energy costs of convergence and illustrates how the design of special-purpose devices can help identify energy scale-down methods in general-purpose devices. As a way of validating our energy scale-down approach, Section 3 proposes three specific scale-down optimizations in the context of the display, wireless, and processor components of the system and shows how they can significantly enhance battery life. Section 4 concludes the paper.

2 Energy Scale-Down

2.1 Energy Costs of Convergence

To validate the hypothesis that special-purpose devices are low-energy devices when compared with converged or general-purpose devices, we measured the energy use of a broad range of devices used for different tasks.

2.1.1 Methodology

We are not aware of any other previous work that has performed such a broad comparison of mobile devices for such a wide range of mobile tasks. Consequently, we have had to make a number of choices when designing our experimental methodology. Of the large design space possible for a study like ours, we chose to focus on quantifying the wide range of energy usage *per task*. We chose our tasks to be representative of the typical activities mobile users would perform. We focused on commercial products optimized for one or more of these tasks.

Devices: Figure 1 summarizes the key characteristics of the devices that we use. The individual devices we consider include a laptop, a handheld, a cell phone, a high end pager, a high-end MP3 player, a low-end MP3 player, and a small "memo" voice recorder. Unless otherwise noted, all the units were set to default settings. The laptop and handheld were set to never turn off. Also, given that the backlight for the handheld chosen is relatively unique in the handheld market (with power consumption atypical of other handhelds in the market), we performed all our experiments with the backlight set to minimum power mode.

Class: Device	CPU	Storage	Display	Wireless	Interfaces	OS
Laptop: Compaq Armada M300	600 MHz Pentium II	256 MB RAM, 12 GB disk	1024x768, 12.1" TFT	Lucent WaveLAN Gold PCMCIA, IEEE 802.11	full-function keyboard, speaker, mic., stereo audio-output jack, and others	Win XP Pro
Handheld: Compaq iPAQ 3630	206 MHz Intel Strong ARM	32 MB RAM, 32 MB ROM	240x320 2.26" TFT display	Same as laptop	touchscreen interface, speaker, microphone, stereo audio-out jack	Pocket PC
Cell phone: Nokia 8260	not in spec	not in spec	73x50, 1.2" x0.8" monochrome LCD+backlight	AT&T wireless	GSM-like headset jack, vibration notification and audio output	Proprietary
High-end pager: Blackberry W1000	Intel 386 (MHz not in spec)	4 MB flash, 512 KB SRAM	8-line (x28 char) LCD+backlight	Tx frequency:896-902 MHz; Rx freq: 935-941MHz	Trackwheel, 31-key qwerty keypad, tone and vibration notification	Proprietary
Low-end MP3: Compaq iPAQ PA-1	not in spec	two 32MB flash card	7x66, 1" x0.4" LCD +backlight	None	Buttons	Proprietary
High-end MP3: Nomad jukebox (DAP-6G01)	not in spec	8MB DRAM, 6GB disk	132x64 LCD+backlight	None	Stereo headphone jack	Proprietary
Voice recorder: Voicelt VT-90	not in spec	not spec (max. record time of 90 sec)	None	None	Buttons	Proprietary

Fig. 1. Devices evaluated in power comparison study

Task	Description
Email	The benchmark tries to capture typical activities associated with an email application. The first component (Rcv) captures the power for receiving messages and the power for the notification events. An automatic script from a remote machine sends out two sets of 10 messages separated by a pause. All volumes are set to maximum and vibrate-mode, if any, is turned on. The second benchmark (Reply) includes aspects of reading, composing, and sending messages.The benchmark models seven forwards and one single-line reply. The messages chosen include a 10KB HTML announcement and a 4KB text message.
MP3	This benchmark measures the power consumed to play the first two minutes of a 6.44MB MP3 song recorded at a bit rate of 192 Kbps. The default Windows Media Player was used to play the song on the laptop and handheld; the high-end and low-end MP3 players had proprietary interfaces to play the song. Power readings were taken with the same set of headphones for all the appliances. Additionally, when available, the power was also measured with the speaker (for the laptop and the handheld). In these cases, the power was measured with the volume set to maximum.
Web browsing	The web browsing benchmark included connecting to an external link with a large number of embedded images. The benchmark refreshes the page once the page is downloaded. All experiments were performed at similar times to minimize network effects.
Text notes taking	This benchmark included the first two minutes of typing in the rules from the table of contents of "The Elements of Style" by Strunk and White, by an operator familiar with the user interface.
Audio notes taking	For this benchmark, we read out loud the first 9 rules from the table of contents of the same book as above one at a time and played back the recordings one at a time.
Two-way messaging	We designed two benchmarks to capture the activities associated with messaging for a mobile user. The first benchmark stresses *instant text messaging* , while the second benchmark stresses *voice chats (or phone conversations)* . In both these cases, we followed a pre-determined script. In order to capture the power consumption of notification events (audio or vibration alerts), we began the conversation with a request into the measured device and then one minute into the messaging, we disconnected the conversation and began another conversation, this time initiated by the measured device. As before, we assumed reasonable skills with the user interfaces, and all volume controls were set to maximum. For the windows environments (laptop and handheld), we used MSN messenger for the text and voice chats.

Fig. 2. Tasks evaluated in power comparison study

Tasks: Figure 2 summarizes the key tasks that we studied. The tasks we consider include email handling (notification, sending and receiving), MP3 song-playing (speaker and headphone), web browsing, notes taking (text and voice), and two-way messaging (text and voice). For each application (a task/device combination), we designed a two-minute long benchmark that we felt represented typical use. In applications where there were elements of non-repeatability, we repeated the experiments several times to ensure that all effects were adequately capture

2.1.2 Measurement Setup

For our experiments we measured total system power. For the current drawn by the device, we measured the voltage across a 0.10 ohm sense resistor (tolerance 1%) between the power source and the device. In order to reduce noise, we amplified the voltage across the resistor, which was then measured by a data acquisition system. For all devices except the cell phone, we removed the batteries and ran the device from a DC power supply, measuring current as it entered the device. Since our cell phone wouldn't run in this manner, we used a fully charged battery with a sense resistor between the cell and the additional electronics we found within the battery case. In order to observe time-varying behavior, we collected a 2 minute trace of each device and application at sample rate of 10,000 samples per second. Traces included both the current (as output by the MAX4127) as well as the power supply voltage of the system under test. Power was computed as the product of the voltage and current samples. In this paper, we report only average numbers in the interests of space.

2.1.3 Measurement Results

Figure 3 summarizes the average power consumption of the devices and benchmarks that we consider. As we expect, we observe that the different devices spend different amounts of power, even when providing similar service. However, more surprising, we see that the variations between these readings are very large, ranging from 950% to over 22,000%. For example, the energy use of our MP3 application varies by a factor of 49 when playing the identical music on different devices. Our email reply benchmark consumes between 71.5mW and 16.26W on different devices; that corresponds to nearly a factor of 227.

	Email		MP3			Notes		Messaging		
Device	Rcv	Reply	Speaker	Headset	Browse	Text	Audio	Text	Audio	Idle
Laptop	15.16 W	16.25 W	18.02 W	15.99 W	16.55 W	14.20 W	14.65 W	14.40 W	15.50 W	13.975 W
Handheld	1.386 W	1.439 W	2.091W	1.700 W	1.742 W	1.276 W	1.557 W	1.319 W	-	1.2584 W
Cellphone	539 mW	472 mW	-	-	-	-	-	392 mW	1147 mW	26 mW
Email Pager	92 mW	72 mW	-	-	-	78 mW	-	-	-	13 mW
High-end MP3	-	-	-	2.977 W	-	-	-	-	-	1.884 W
Low-end MP3	-	-	-	327 mW	-	-	-	-	-	143 mW
Voice Recorder	-	-	-	-	-	-	166 mW	-	-	17 mW
variance	*16496%*	*22727%*	*861%*	*4890%*	*950%*	*18252%*	*8825%*	*3673%*	*1351%*	*107500%*

Fig. 3. Power consumption for special-purpose and general-purpose mobile devices

2.2 The Need for Energy Scale-Down

In all the cases in the previous section, the large variances were primarily attributable to the difference between the low energy consumed by an application-specific device optimized for energy and the high-energy consumed by a general-purpose device optimized for functionality. The energy differences can be largely explained in terms of the individual components in each system. Focusing on one specific example of the email application, when moving from a laptop (highest power) to an email pager (lowest power), a number of components are replaced with less powerful components. This provides lower power and lower, or perhaps different, functionality (but optimized for the characteristics of the application and market acceptance of reduced features). The display and CPU are scaled down, and the wireless system has different characteristics. The application software is scaled down to provide just the essential features.

The scale-down of the software is particularly interesting, since it often tends to be ignored. In our example, the software in these benchmarks includes Outlook in the laptop, Pocket Outlook on the iPAQ, and simple text email software on the cell phone and the pager. Each of these provides different functionality. Unlike a study of, say, CPU performance, where we would like to keep the benchmarks constant to ensure a fair comparison, we argue that when comparing power consumption of different implementations of tasks, software is an important component that also needs to be scaled to meet the user' s desired functionality.

Below, we discuss the power numbers and how they relate to our hypothesis that devices with general-purpose or combined functionality consume more power because they do not provide the adaptivity to respond to application requirements. In these discussions, we highlight examples of how the optimizations found in special-purpose devices can be useful in improving energy efficiency for our general-purpose devices, in the context of one application - email.

Email application on different mobile devices: Comparing the cell phone and the email pager device for the email benchmark, we observe an interesting trend with the pager having a factor of 6 lower power in spite of its larger (and hence potentially higher power) display. An examination of the traces indicates that that the pager's wireless system has significantly lower activity compared to the cell phone. The cell phone demonstrates the compromises of convergence, even on small special-purpose devices. Like other handheld devices, the wireless protocol for a cell phone device typically leaves the radio off most of the time, turning it on periodically to check for activity. Our traces of both the phone and pager show periodic energy spikes that we hypothesize are the radio waking up. On the phone, these spikes are approximately 1.2 seconds apart. This corresponds to adding an average ring latency of 0.6 seconds to each incoming phone call. On the pager, these spikes are approximately 5 seconds apart, leading to an average 2.5 second latency on incoming email (a factor of 4 compared to the cell phone). Both these latency numbers seem appropriate to the application at hand, and this variation in the wakeup period can be considered another example of component scale-down for better energy efficiency. Thinking in terms of scale-down, we might consider designing a cell phone protocol in a way that allows us to set the phone to an email-only mode,

allowing us to lengthen the average latency to 2.5 seconds, approaching the energy consumption of an email pager.

Comparing the cell phone against the handheld shows an additional energy cost for additional generality. The handheld email benchmark consumes approximately 3 times the energy as the cell phone. The wireless system for the handheld, in particular, is optimized for low latency local communication at a high bit rate, rather than wide-area communication at a bit-rate tuned for speech. The ideal scalable handheld would incorporate both types of wireless systems or (in an ideal world) a single system that could scale its features to match either wireless system. Even without changing the range or bandwidth, however, one scale-down approach would be to change the latency requirements of the wireless system to match the task at hand. A typical wakeup period for 801.11b is 100ms, while 5 seconds would be fine for email. Even though the 802.11b protocol supports longer wakeup periods, software generally does not provide the necessary interfaces for taking advantage of this. A software system designed for scale-down of energy would provide these interfaces, and applications would facilitate scaling the components to match the requirements of the task at hand.

In addition to the wireless component, we can observe opportunities for scale down in other components of the system. One important component of the handheld is the display which has no scale-down capacity. Even if the users are comfortable with a smaller, lower-resolution, lower-color screen to scroll through when reading their messages, there are no software or hardware interfaces to support this. This problem is particularly exacerbated in the laptop with its emphasis on a much larger form-factor. Similarly, the processor component of the various devices varies significantly in power. The rated power for the Pentium II and StrongARM processors used in the general-purpose device is orders of magnitude higher than the rated power of the embedded processors used in the special-purpose devices, even though for some of our tasks, similar computation is performed on all the processors.

Summary: Though we focused on one application in the interests of space, similar trends are present in the other applications as well. For all the tasks, our results show that the special-purpose devices have orders of magnitude lower power consumption compared to the general-purpose devices, validating our intuition that they could serve as good examples of energy use that general-purpose devices may aim for. We suggest that researchers, in part, evaluate the success of their energy scale down efforts by how closely they approach the energy consumption of such application specific devices. For example, in our ideal world, a laptop playing an MP3 file could, if the user desired, consume no more power than the best available MP3 player. The next section discusses specific energy scale-down optimizations that work towards this goal.

3 Energy Scale-Down Optimizations

As a way of closing the gap between the special-purpose and general-purpose devices, we suggest integrating the notion of *requirement-specific energy scale-down* at all

levels of the system, namely providing for the *design and use of adaptivity in hardware and software* to exploit mismatches between system functionality and workload/user requirements. Specifically, we suggest considering each component in the general-purpose device and comparing it to the requirements of the applications using that device. Ideally, each general-purpose component should be capable of scaling down its energy use to match the design point used by the application with the lowest requirements. There are two alternatives for achieving this: *gradation-based scaling*, where the component has a wide range of adaptability, or *plurality-based scaling*, where the device chooses among multiple components with different properties. Below, we discuss three example scale-down optimizations in the context of the display, wireless, and processor components of the system that attempt to use adaptivity to improve the efficiency of energy use in the device.

3.1 Display Scale-Down

The user acceptance of smaller, lower-quality, and lower-energy displays in special-purpose devices indicates that certain tasks may not always need the most aggressive functionalities of the display (e.g., large size, full color, great resolution, backlight, etc.). In contrast, most current general-purpose systems include *"one-size-fits-all"* displays targeted at the needs of the most aggressive workload/user. This can lead to large energy-inefficiencies in the display energy consumption of other workloads and users. This motivates the need for *energy-adaptive display systems* [IyerLuo+2003] that consume energy only on portions and characteristics of the screen that the user considers relevant.

To understand user needs for displays, we analyzed the display usage traces from 17 users, representing a few hundred hours of active screen usage. Our user population covered a cross section of mobile system usage (administrative tasks, code development, personal productivity, entertainment, etc.). We found that on average, the window of focus – a good first-order indication of the area of interest to the user – uses only about 60% of the total screen area. Additionally, in many cases, the screen usage is associated with content that could have been equivalently displayed, with no loss in visual quality, on much simpler lower power displays. Our analysis of the user traces indicated that many of these mismatches could be traced back to the typical content of the windows as opposed to specific user preferences. For example, windows with low content (email composition, terminals, system status and control messages, menu widgets, etc.) were the dominant types of smaller-sized windows and windows with relatively higher content (web browsing, code development, PowerPoint, document reading, etc.) were the dominant types of larger-sized windows.

Based on these observations, we evaluated a family of display scale-down optimizations built on emerging OLED display technologies [Stanford2001] that allow the energy consumption to be proportional to the overall light output of the display. At a software level, we designed *energy-aware user interfaces* that change the luminescence and color of the non-active screen areas to reduce power while leaving the active screen area (the window of focus) unchanged. Some examples of the interfaces presented to the user are summarized in Figure 4. Our experience with prototypes in

dictates broad acceptance of these interfaces among users, particularly in the context of longer battery life. Figure 5 summarizes the energy benefits from applying the fully-dimmed optimization. Since the energy benefits are a function of the screen background and the window background colors, in addition to the default windows configuration (teal screen background and white windows background) we bracketed our results by evaluating other configurations. As the results indicate, integrating scale-down optimizations in the display design can achieve factors of 1.5-5 reduction in the display energy. The benefits for individual users vary, up to 10X in some cases.

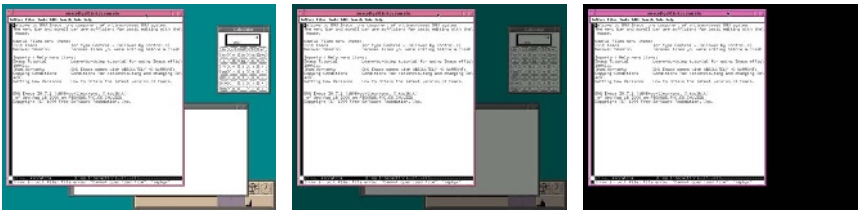


Fig. 4. Display interfaces with energy scale-down

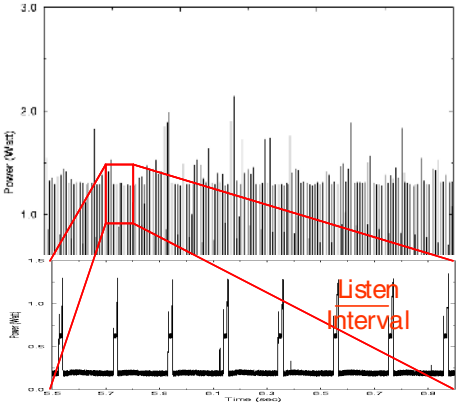


Fig. 5. Wireless power consumption from beacons

In addition to the specific family of designs that we evaluated, energy scale-down can be integrated in other ways as well. For example, with display technologies that do not support energy variability, a hierarchy of displays or alternative communication and user input methods could be used to provide the hardware support for adaptivity. Similarly at the software level, we can extend the optimizations discussed above to include pointer-based user-relevance determination, time-based dimming interfaces, as well as intra-application support for adaptivity. Though not presented

here in detail, the power benefits and user acceptance of energy-adaptive display interfaces for handhelds was also studied. Again, the results show factors of 1.3 to 8.3 with high user acceptance ratings [Harter+2003].

3.2 Wireless Scale-Down

As discussed earlier, special-purpose communication devices such as cell phones or pagers consume significantly lower energy in the wireless system by minimizing the activity on the wireless network. In contrast, the general-purpose systems often have wireless subsystems and protocols optimized for the most aggressive connection requirements (e.g., bandwidth, latency, response) leading to energy inefficiencies. The wireless scale-down optimization discussed in this section addresses this drawback.

To understand the workload's requirements for the wireless system, we evaluated the energy spent in the 802.11b wireless sub-system of a general-purpose mobile device for an email application [AbouGhazalaMayo+2003]. Our results indicated that a large fraction of the wireless energy was spent when the system was in idle mode (as opposed to transmitting/receiving messages). Further, as indicated in Figure 6, the power consumed in the idle mode was dominated by the power spent in listening for periodic beacons to ensure timely response to incoming transmissions. Typical default configurations set the "listen interval" (the time between beacons) to be 100 ms. In contrast, the mean time period between message receipts for our representative user was many minutes.

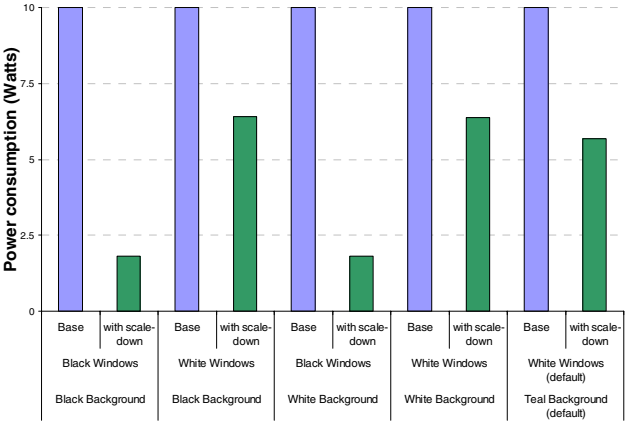


Fig. 6. Wireless power consumption from beacons

Based on these observations, we implemented an energy-adaptive wireless system that could adapt its listen interval to better respond to the desired application response times. For example, if the user finds an email latency of 5 seconds to be

acceptable, this information should be reflected in the listen interval of the wireless protocol. We evaluated two approaches. For short increases in listen interval (5 seconds, *"Listen change"* in Figure 7), we changed the listen interval by changing the corresponding parameter in the 802.11 protocol. For longer increases (60 seconds, *"Listen shut"* in Figure 7), we turned off and restarted the wireless card. As shown in the Figure, the wireless scale-down optimizations achieve factors of 1.3 to 9 better energy consumption.

Though the specific optimization considered above is fairly straightforward, the same insights can be applied to other configurations, particularly in the context of multiple wireless networks in the same device. In particular, a small

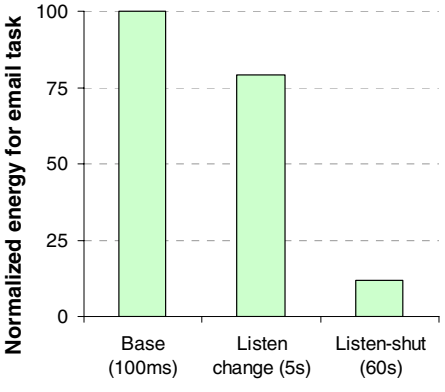


Fig. 7. Wireless scale-down benefits

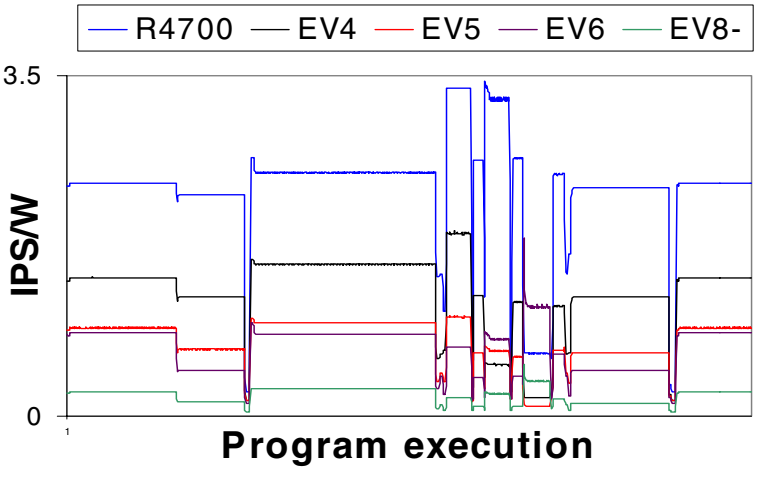


Fig. 8. Energy efficiency differences between processor cores

amount of additional hardware in the form of a small low-power radio to supplement the main wireless network can provide even finer-granularity of adaptivity [ShihBahl+2002].

3.3 Processor Scale-Down

The third scale-down optimization that we consider focuses on the processor component of the system power and is motivated by the observation that in some cases, a lower power and lower functionality processor is often enough to adequately perform a particular task. Once again, the general-purpose system includes a processor that is typically targeted at the most aggressive workload requirements (performance) and does not have a simple mechanism to scale down to the lower functionalities required by other tasks.

Fourteen integer and floating-point applications from the SPEC2000 benchmark suite were simulated on five different processor cores supporting the same instruction set architecture [KumarFarkas+2003] [KumarFarkas+2003b]. Figure 8 shows the energy efficiency of the five cores over the course of execution of a representative benchmark. The five cores approximate the MIPS R4700, the Alpha 20164 (EV4), 21164 (EV5), 21264 (EV6) and a potential next-generation approximation to the EV6 (EV8-). The results indicated that different processor cores have different energy efficiencies based on the nature of the workload being executed on them.

Based on the analysis, a plurality-based scale-down optimization for processors based on heterogeneous single-ISA multi-core architectures may yield energy benefits. The key idea is to have the main high-performance processor supplemented with other satellite processors that span the power-performance design space. The workload is run on the core with the best energy efficiency properties for that workload, and the other cores are shut off. This should be possible with little additional die area, as the die size of such a combined system is little more than the size of the largest processor, as shown in Figure 9. Evaluation of both static and dynamic heuristics for workload migration indicate an average energy improvement of 1.4X (factors of 2 to 10 in six of the applications) with less than 3% speed degradation. In cases when lower performance is acceptable, it is possible to achieve factors of 3 to 11 reduction in energy with less than 25% loss in performance[KumarFarkas+2003][KumarFarkas+2003b].

Though limited to the specific technology implementation of the processor, another potential scale-down optimization for processors is the use of voltage and frequency scaling [PillaiShin2001]. Additional scale-down optimizations can also consider the use of

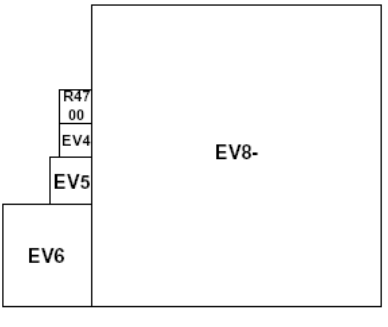


Fig. 9. CPU scale-down architecture

energy-aligned cores that design the architecture of the specific cores to better improve energy efficiency.

4 Conclusions

As the mobile device market matures, the large number of mobile computing devices optimized for different form factors and functionalities is likely to be replaced with a few commonly-accepted devices that integrate multiple functionalities in the same device. Indeed, this is evidenced by the large number of announcements about "combination products" such as camera/cell phones or cell phones/personal organizers or gaming devices/MP3 players, etc. However, while there has been a great focus on designing these devices to scale up the device properties to provide the greatest of the various functionalities, there has been very little work in providing approaches to scale-down the device to the least of the various functionalities. This has particularly been a problem in the case of the energy consumption of these devices when a system component consumes more energy by virtue of supporting a larger function set than what is desired by an application (E.g., reading black and white text messages on a color screen cell phone organizer).

This paper argues that along with the importance given to scaling up functionality, equal importance should be given to designing methods in hardware and software to *scale down* the energy. Individual applications or users can then use these mechanisms to control the energy based on their specific requirements. As validation of this thesis, we compare the energy consumption of general-purpose devices (that support the function set required by several tasks) with special-purpose devices targeted specifically at particular tasks. Across the range of tasks we considered – sending and receiving email, web browsing, listening to MP3 music, text and audio notes taking, and text and phone messaging – we observed inefficiencies in the general-purpose devices that led to factors of 10 to 100 higher energy consumption compared to the special-purpose devices. To the best of our knowledge, ours is the first such study to perform a consistent comparison of the energy consumption of the various devices. Furthermore, an analysis of the differences between the devices illustrates opportunities when user requirements can be met with much lower energy use.

Building on this analysis, we proposed and evaluated three specific scale-down approaches that exploit an awareness of the user and task requirements to scale down the energy selectively. In the first case, the system leverages the observation that users typically use only a fraction of their screen area and selectively controls the pixel intensity on the screen to match the power consumed in the display with the portions relevant to the user. In the second case, the system leverages the observation that users are willing to tolerate longer response times than what is currently provided by wireless networks and by exposing this tolerance to the protocol, reduces the power consumed in the wireless system. The third case observes that different processor designs are better matched, from an energy efficiency point of view, to different workloads and uses a multi-core architecture to reduce energy. In all these cases, the energy scale-down optimizations achieve close to a factor of 2 to 10 better energy consumption compared to existing methods of designing systems.

While the three optimizations we consider in the paper validate our argument on the potential of energy scale down in future designs, we believe that we have only scratched the surface. Previously proposed means for adaptivity can complement these to provide further scale-down, for example, voltage and frequency scaling [PillaiShin2001], architectural gating [ManneKlauser+1998], selective memory usage [LebeckFan+2000] and disk spin-down [DouglisKrishnan+1994]. However, the factors of 10 to 100 indicated in our energy comparison show that we still have a significant potential in terms of energy savings to attain. Additional mechanisms for adaptivity in hardware and software for energy scale down and policies for requirements-aware use of this adaptivity will be essential as we try to further address the battery life challenges in future systems.

Acknowledgements

We would like to thank the input and support of Keith Farkas, Subu Iyer, Norm Jouppi, Rakesh Kumar, Annie Luo, Jim Rowson, John Sontag, and Dean Tullsen. We would also like to thank the reviewers for their comments.

References

[AbouGhazalaMayo+2003] Nevine Abou-Ghazala, Robert Mayo, and Parthasarathy Ranganathan. Idle Mode Power Management for Personal Wireless Devices, *Hewlett Packard Technical Report HPL 2003-102*, 2003

[DouglisKrishnan+1994] Fred Douglis, P. Krishnan and B. Marsh. Thwarting the power hungry disk. *Proceedings of the 1994 USENIX Conference*, pp 293-306, January 1994

[Harter+2003] Timoth Harter, Sander Vroegindeweij, Erik Geelhoed, Meera Manahan, and Parthasarathy Ranganathan. Energy-aware User Interfaces: An Evaluation of User Acceptance, Proceedings of the Conference on Human Factors in Computing Systems (CHI), April 2004

[Hamburgen+1989] Bill Hamburgen, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, Joel Bartlett. Characterization of Organic Illumination Systems. *Hewlett Packard Technical Report WRL-TN-13*, 1989

[IyerLuo+2003] Subu Iyer, Annie Luo, Robert Mayo, and Parthasarathy Ranganathan. Energy-Adaptive Display System Designs for Future Mobile Environments. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, 2003, pp245-258

[KumarFarkas+2003] Rakesh Kumar, Keith Farkas, Norman Jouppi, Parthasarathy Ranganathan, and Dean Tullsen. Processor Power Reduction via Single-ISA Heterogeneous Multi-core Architectures. *Computer Architecture Letters*, Volume 2, April 2003

[KumarFarkas+2003b] Rakesh Kumar, Keith Farkas, Norman Jouppi, Parthasarathy Ranganathan, and Dean Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction, *Proceedings of the 36th International Symposium on Microarchitecture, December 2003*

[LebeckFan+2000] A. Lebeck, X. Fan, H. Zeng, and C.S.Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, November 2000

[ManneKlauser+1998] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp 132-141, June 1998

[PillaiShin2001] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th Symposium on Operating System Principles*. 2001.

[ShihBahl+2002] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *Proceedings of the Eighth Annual ACM Conference on Mobile Computing and Networking*, Atlanta, Georgia, USA, September 2002.

[Stanford2001] Stanford Resources Inc. *Organic Light Emitting Diode Displays: Annual Display Industry Report*, Second Edition 2001

Efficient Scratchpad Allocation Algorithms for Energy Constrained Embedded Systems

Manish Verma, Lars Wehmeyer, and Peter Marwedel

Department of Computer Science XII,
University of Dortmund, 44221 Dortmund, Germany
{Manish.Verma, Lars.Wehmeyer, Peter.Marwedel}@uni-dortmund.de

Abstract. In the context of portable embedded systems, reducing energy is one of the prime objectives. Memories are responsible for a significant percentage of a system's aggregate energy consumption. Consequently, novel memories as well as novel memory hierarchies are being designed to reduce the energy consumption. Caches and scratchpads are two contrasting variants of memory architectures. The former relies completely on hardware logic while the latter requires software for its utilization. Most high-end embedded microprocessors today include onchip instruction and data caches along with a scratchpad.

Previous software approaches for utilizing scratchpad did not consider caches and hence fail for the prevalent high-end system architectures. In this work, we use the scratchpad for storing instructions. We solve the allocation problem using a greedy heuristic and also solve it optimally using an ILP formulation. We report an average reduction of 20.7% in instruction memory energy consumption compared to a previously published technique. Larger reductions are also reported when the problem is solved optimally.

The scratchpad in the presented architecture is similar to a preloaded loop cache. Comparing the energy consumption of our approach against that of preloaded loop caches, we report average energy savings of 28.9% using the heuristic.

Keywords: Memory architectures, Memory allocation, Energy aware compilation, Integer Linear Programming, Memory energy modeling.

1 Introduction

Over the past decade, the popularity of mobile embedded devices such as mobile phones, digital cameras etc. has been one of the major driving forces in technology. The computing power of early desktop computers is now available in a handheld device. Unfortunately, battery technology could not keep pace with the advances made in silicon technology. As a result, contemporary mobile embedded systems suffer from limited battery capacity. Reduced energy consumption translates to reduced dimensions, weight and cost of the device. In such a competitive market, these reductions might be sufficient to provide an edge over competing products.

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 41–56, 2004.

Several researchers [4, 16] have identified the memory subsystem as the energy bottleneck of the entire system. In fact, fetches from the instruction memory typically account for much of a system's power consumption [10]. Memory hierarchies are being introduced to reduce the memory system's energy dissipation. Caches and scratchpad memories represent two contrasting memory architectures. Caches improve performance by exploiting the available locality in the program. As a consequence, energy consumption is also reduced. However, they are not an optimal choice for energy constrained embedded systems. Caches, apart from the actual memory, consist of two additional components [22]. The first component is the tag memory required for storing information regarding valid addresses. The second component is the hardware comparison logic to determine cache hits and cache misses. These additional components consume a significant amount of energy per access to the cache irrespective of whether the access translates to a hit or a miss. Also, caches are notorious for their unpredictable behavior [14].

On the other end of the spectrum are the scratchpad memories, consisting of just data memory and address decoding circuitry. Due to the absence of tag memory and comparators, scratchpad memories require considerably less energy per access than a cache. In addition, they require less onchip area and allow tighter bounds on WCET prediction of the system. However unlike caches, scratchpads require complex program analysis and explicit support from the compiler. In order to strike a balance between these contrasting approaches, most of the high-end embedded microprocessors (e.g. ARM10E [1], ColdFire MCF5 [15]) include both onchip caches and a scratchpad.

We assume a memory hierarchy as shown in figure 1.(a) and utilize the scratchpad for storing instructions. The decision to store only instructions is motivated by the fact that the instruction memory is accessed on every instruction fetch and the size of programs for mobile embedded devices is smaller compared to their data size requirements. This implies that small scratchpad memories can achieve greater energy savings when they are filled with instructions rather than with data. In this paper, we model the cache behavior as a conflict graph and allocate objects onto the scratchpad considering their effect on the I-cache. As shown later, the problem of finding the best set of objects to be allocated on the scratchpad can be formulated as a non-linear optimization problem. Under simplifying conditions, it can be reduced to either a Weighted Vertex Cover [9] problem or a Knapsack [9] problem, both of which are known to be NP-complete problems. A greedy heuristic is used to solve the scratchpad allocation problem. An optimal solution is also obtained by formulating the scratchpad allocation problem as an ILP problem. We compare our approach against a published technique [19]. Due to the presence of an I-cache in our architecture, the previous technique fails to produce optimal results and may even lead to the problem of *cache thrashing* [11].

We also compare our approach to that of preloaded loop caches [10], as the utilization of the scratchpad in the current setup (see figure 1) is similar to a loop cache. Preloaded loop caches are architecturally more complex than scratchpads,

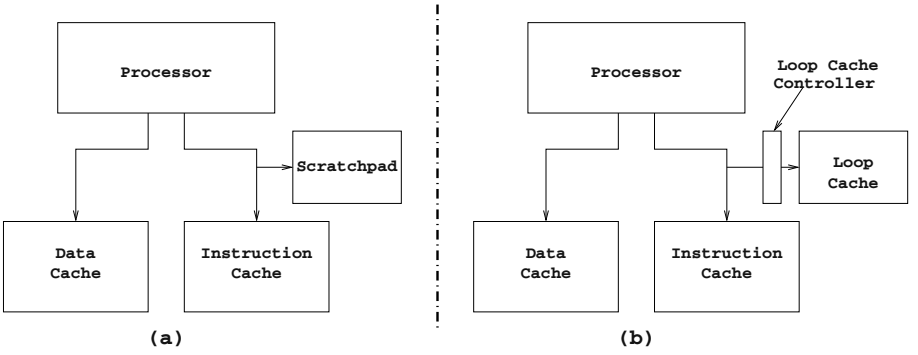


Fig. 1. System Architecture: (a) Scratchpad (b) Loop Cache

but are less flexible as they can be preloaded with only a limited number of loops. We demonstrate that using our allocation algorithm, scratchpad memories can outperform their complex counterparts.

In the next section, we describe related work and detail the shortcomings of the previous approaches. Section 3 describes the information regarding memory objects, cache behavior and the energy model. Section 4 presents the scratchpad allocation problem in detail, followed by the description of the proposed heuristic and the ILP formulation. The experimental setup is explained in section 5. In section 6 we present the results for an ARM based system and end the paper with a conclusion and future work.

2 Related Work

Analytical energy models for memories [12] have been found to be fairly accurate. We use *cacti* [22] to determine the energy per access for caches and preloaded loop caches. The energy per access for scratchpad memories was determined using the model presented in [3].

Application code placement techniques [17, 21] were developed to improve the CPI (cycles per instruction) by reducing the number of I-cache misses. Those basic blocks that are frequently executed in a contiguous way are combined to form so-called traces [17]. Authors in [17] placed traces within functions, while [21] placed them across function boundaries to reduce the I-cache misses.

Several researchers [2, 16] have utilized scratchpad memories for assigning global/local variables, whereas only Steinke et al. [19] considered both program and data parts (memory objects) to be allocated onto the scratchpad. They assumed a memory hierarchy composed of only scratchpad and main memory. Profit values were assigned to program and data parts according to their execution and access counts, respectively. They then formulated a knapsack problem to determine the best set of memory objects to be allocated to the scratchpad memory.

Though this approach is sufficiently accurate for the used memory hierarchy, it is not suitable for the current setup. The assumption that execution (access) counts are sufficient to represent energy consumption by a memory object fails in the presence of a cache, where execution (access) counts have to be decomposed into cache hits and misses. The energy consumption of a cache miss is significantly larger than that of a cache hit. Consequently, two memory objects can have the same execution (access) counts, yet have substantially different cache hit/miss ratio and hence energy consumption. This discussion stresses the need for a more detailed energy model taking these effects into account. In addition, maintaining the conflict relationships between memory objects is not considered during code placement using the previous approach. The memory objects are moved instead of copying them from main memory to the scratchpad. As a result, the layout of the entire program is changed, which may cause completely different cache access patterns and thus lead to erratic results.

Authors in [13] proposed an instruction buffer to act as an alternative location for fetching instructions in order to improve the energy consumption of a system. Loops identified by the short backward branch at the end of the first iteration are copied to the instruction buffer during the second iteration. From the third iteration onwards, instructions are fetched from the instruction buffer instead of the L1 I-cache, given that no *change-of-flow* (e.g. branch) statements are contained within the loop. Ross et al. [10] proposed a Preloaded Loop Cache which can be statically loaded with pre-identified memory objects. Start and end addresses of the memory objects are stored in the controller, which on every instruction fetch determines whether to access the loop cache or the L1 I-cache. Consequently, the loop cache can be preloaded with complex loops as well as functions. However, to keep the energy consumption of the controller low, only a small number of memory objects (typically 2-6) can be preloaded.

The problem of being able to store only a fixed number of memory objects in the loop cache will lead to problems for large programs with several hot spots. As in [19], memory objects are greedily selected only on the basis of their execution time density (execution time per unit size). In the wake of the discussion we enumerate the contributions of this paper.

- It for the first time studies the combined effect of a scratchpad and an I-cache on the memory system's energy consumption.
- It stresses the need for a sophisticated allocation algorithm by demonstrating the inefficiency of previous algorithms when applied to the present architecture.
- It presents a novel scratchpad allocation algorithm which can be easily applied to a host of complex memory hierarchies.
- It demonstrates that scratchpad memories together with an allocation algorithm can replace preloaded loop caches.

Please note that in the rest of this paper, energy consumption refers to the energy consumption of the instruction memory subsystem. In the following section, we describe preliminary information required for understanding our approach.

3 Preliminaries

We start by describing the assumed architecture for the current research work, followed by the description of the memory objects. The interaction of memory objects within the cache is represented using a conflict graph, which forms the basis of the proposed energy model and the algorithm.

3.1 Architecture

For the presented research work we assume a Harvard architecture (see figure 1(a)) with the scratchpad at the same horizontal level as the L1 I-cache. The scratchpad is mapped to a region in the processor's address space and acts as an alternative non-cacheable location for fetching instructions. As shown in figure 1(b), the preloaded loop cache setup is similar to using a scratchpad.

3.2 Memory Objects

In the first step of our approach, memory objects within the program code are identified. The memory objects are then distributed between offchip main memory and non-cacheable scratchpad memory to minimize energy consumption. The well known compiler optimization *trace generation* is used to identify the memory objects. A *trace* is a frequently executed straight-line path, consisting of basic blocks connected by fall-through edges [21]. Dynamic profiling is required to determine traces in the program. Our traces are kept smaller than the scratchpad size, as larger traces can not be placed onto the scratchpad as a whole. The traces are appended with NOP instructions to align them to cache line boundaries. This ensures a *one-to-one relationship* between cache misses and corresponding traces. The rational behind using traces is threefold. Firstly, traces improve the performance of both the cache and the processor by enhancing the spatial locality in the program code. Secondly, due to the fact that traces always end with an unconditional jump [21], they form an atomic unit of instructions which can be placed anywhere in memory without modifying other traces. Finally, traces are accountable for every cache miss caused by them. In the rest of the paper, unless specified, traces will be referred to as memory objects (MO). In the following subsection, we represent the cache behavior at the granularity of memory objects by a conflict graph.

3.3 Cache Behavior (Conflict Graph)

The cache maps an instruction to a cache line according to the following function:

$$Map(address) = address \ \textbf{mod} \ \frac{CacheSize}{Associativity * WordsPerLine}$$

Similarly, a memory object is mapped to cache line(s) depending upon its start address and size. Two memory objects potentially cause a conflict in the cache

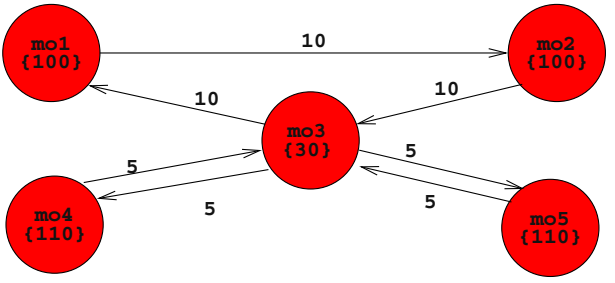


Fig. 2. Conflict Graph

if they are mapped to at least one common cache line. This relationship can be represented by a conflict graph G (see figure 2), which is defined as follows:

Definition: The *Conflict Graph* $G = (X, E)$ is a directed weighted graph with node set $X = \{x_1, \ldots, x_n\}$. Each vertex x_i in G corresponds to a memory object (MO) in the application code. The edge set E contains an edge e_{ij} from node x_i to x_j if a cache-line belonging to x_j is replaced by a cache-line belonging to x_i using the cache replacement policy. In other words, $e_{ij} \in E$ if there occurs a cache miss of x_i due to x_j. The weight m_{ij} of the edge e_{ij} is the number of cache lines that need to be fetched if there is a miss of x_i that occurs due to x_j. The weight f_i of a vertex x_i is the total number of instruction fetches within x_i.

In order to build up the conflict graph for a program, we first need to identify the memory objects to be considered by our algorithm. We use profiling to determine traces. In order to mark the vertices with the total number of instruction fetches and to determine the number of conflict misses among the memory objects, dynamic profiling is also required. The determined values are then attributed to vertices and conflict edges, respectively. In order to minimize the influence of the chosen input data set on the results, average values generated by using several distinct input vectors can be used.

The conflict graph as shown in figure 2 is a directed graph because the conflict relationship is *antisymmetric*. The conflict graph G and the energy values are utilized to compute the energy consumption of a memory object according to the energy model proposed in the following subsection.

3.4 Energy Model

As mentioned before, all energy values refer to the energy consumption of the instruction memory subsystem. The energy $E(x_i)$ consumed by an MO x_i is expressed as:

$$E(x_i) = \begin{cases} E_{SP}(x_i) & \text{if } x_i \text{ is present on scratchpad} \\ E_{Cache}(x_i) & \text{otherwise} \end{cases} \quad (1)$$

where E_{Cache} can be computed as follows:

$$E_{Cache}(x_i) = Hit(x_i) * E_{Cache\_hit} + Miss(x_i) * E_{Cache\_miss} \tag{2}$$

where functions $Hit(x_i)$ and $Miss(x_i)$ return the number of hits and misses, respectively, while fetching the instructions of MO x_i. E_{Cache_hit} is the energy of a hit and E_{Cache_miss} is the energy of a miss in one line of the I-cache.

$$Miss(x_i) = \sum_{x_j \in N_i} Miss(x_i, x_j) \quad \text{with} \tag{3}$$
$$N_i = \{x_j \ : \ e_{ij} \in E\}$$

where $Miss(x_i, x_j)$ denotes the number of conflict cache misses of MO x_i caused due to conflicts with MO x_j. The sum of the number of hits and misses is equal to the number of instruction fetches f_i in an MO x_i:

$$f_i = Hit(x_i) + Miss(x_i) \tag{4}$$

For a given input data set, the number of instruction fetches f_i within an MO x_i is a constant and is independent of the memory hierarchy. Substituting the terms $Miss(x_i)$ from equation (3) and $Hit(x_i)$ from equation (4) in equation (2) and rearranging derives the following equation:

$$E_{Cache}(x_i) = f_i * E_{Cache\_hit} + \tag{5}$$
$$\textstyle\sum_{x_j \in N_i} Miss(x_i, x_j) * (E_{Cache\_miss} - E_{Cache\_hit})$$

The first term in equation (5) is a constant while the second term, which is variable, depends on the overall program code layout and the memory hierarchy. We would like to point out that the approach [10] only considered the constant term in its energy model. Consequently, the authors could not optimize the overall memory energy consumption.

Since there are no misses when an MO x_i is present in the scratchpad, we can deduce the following energy equation:

$$E_{SP}(x_i) = f_i * E_{SP} \tag{6}$$

where E_{SP} is the energy per access of the scratchpad.

4 Problem Description

Once we have created the conflict graph G annotated with vertex and edge weights, the energy consumption of memory objects can be computed. Now, the problem is to select a subset of memory objects which minimizes the number of conflict edges and the overall energy consumption of the system. The subset is bounded in size by the scratchpad size.

In order to formally describe the algorithm we need to define a number of variables. The binary variable $l(x_i)$ denotes the location of memory object x_i in the memory hierarchy:

$$l(x_i) = \begin{cases} 0, \text{ if } x_i \text{ is present on scratchpad} \\ 1, \text{ otherwise} \end{cases} \tag{7}$$

Since a memory object allocated to the scratchpad does not conflict with other memory objects, we can represent $Miss(x_i, x_j)$ (see above) as follows:

$$Miss(x_i, x_j) = \begin{cases} 0, & \text{if } x_j \text{ is present on scratchpad} \\ m_{ij}, & \text{otherwise} \end{cases} \tag{8}$$

where m_{ij} is the weight of the edge e_{ij} connecting vertex x_i to x_j. Function $Miss(x_i, x_j)$ can be reformulated using the location variable $l(x_j)$ and represented as:

$$Miss(x_i, x_j) = l(x_j) * m_{ij} \tag{9}$$

Similarly, the location variable $l(x_i)$ can be used to reformulate the energy equation (1) denoting the energy consumed by the memory object.

$$E(x_i) = [1 - l(x_i)] * E_{SP}(x_i) + l(x_i) * E_{Cache}(x_i) \tag{10}$$

We substitute the energy equations for E_{Cache} and E_{SP} from equations (5) and (6), respectively, into the above equation. By rearranging the terms we transform equation (10) into the following form.

$$\begin{aligned} E(x_i) = f_i * E_{SP} + f_i * [E_{Cache\_hit} - E_{SP}] * l(x_i) + \\ [E_{Cache\_miss} - E_{Cache\_hit}] * [\textstyle\sum_{j \in N_i} l(x_j) * l(x_i) * m_{ij}] \end{aligned} \tag{11}$$

We find the last term is a quadratic degree term, since the number of misses of a memory object x_i not only depends upon its location but also upon the location of the conflicting memory objects x_j.

The objective function E_{Total} denoting the total energy consumed by the system needs to be minimized.

$$E_{Total} = \sum_{x_i \in X} E(x_i) \tag{12}$$

Minimization of the objective function is to be performed while conforming to the scratchpad size constraint.

$$\sum_{x_i \in X} [1 - l(x_i)] * S(x_i) \leq ScratchpadSize \tag{13}$$

The size $S(x_i)$ of memory object x_i is computed without considering the appended NOP instructions. These NOP instructions are stripped away from the

Greedy-Heuristic(G(X,E), ScratchpadSize)
1 Rem_SPSize = ScratchpadSize
2 L = NIL
3 **while** ($\exists x \in X : S(x) \leq$ Rem_SPSize)
4 **do** select $x_i \in X : S(x_i) \leq$ Rem_SPSize $\wedge$
$E(x_i) > E(x_k)\ \forall x_k \in X : S(x_k) \leq$ Rem_SPSize
5 $X = X - \{x_i\}$
6 $E = E - \{e_{ij} | \forall j : j \in N_i\} - \{e_{ji} | \forall j : i \in N_j\}$
7 Rem_SPSize = Rem_SPSize - $S(x_i)$
8 $L = L \cup \{x_i\}$
9 **return** L

Fig. 3. Greedy Heuristic for Scratchpad Allocation Problem

memory objects prior to allocating them to the scratchpad. The non-linear optimization problem can be solved to obtain a scratchpad allocation optimized with respect to energy.

Our problem formulation can be easily extended to handle complex memory hierarchies. For example, if we had more than one scratchpad at the same horizontal level in the memory hierarchy, then we only need to repeat inequation (13) for every scratchpad. An additional constraint ensuring that a memory object is assigned to at most one scratchpad is also required.

The above optimization problem is related to two NP-complete problems viz. Weighted Vertex Cover [9] and Knapsack problem [9]: Under the simplifying assumption that the cache present in the system is large enough to hold all the memory objects without causing a single conflict miss, the energy consumption of a memory object becomes independent of other memory objects. Under this assumption, the problem is reduced to a Knapsack problem with each node having constant weights. On the other hand, if we assume that the energy of an access to the scratchpad E_{SP} is equal to the energy of a cache hit E_{Cache_hit}, equation (11) transforms to the following form and the problem is reduced to the Weighted Vertex Cover problem:

$$E(x_i) = f_i * E_{SP} + [E_{Cache\_miss} - E_{Cache\_hit}] * [\sum_{j \in N_i} l(x_j) * l(x_i) * m_{ij}] \quad (14)$$

Fortunately, approximation algorithms can be employed to obtain near-optimum solutions in polynomial time. In the following section, we will present a greedy heuristic which solves the scratchpad allocation problem near-optimally in most cases. We will also solve the problem optimally using an *Integer Linear Programming* (ILP) based approach.

4.1 Greedy Heuristic

The proposed greedy heuristic tries to put maximum weighted nodes on the scratchpad. It takes as input the conflict graph and the scratchpad size and returns the list of memory object to be allocated onto the scratchpad. The heuristic is formally presented in figure 3.

The heuristic iteratively computes the energy consumption of each memory object which can be placed on the scratchpad memory, considering not only execution counts but also the number of conflict cache misses caused by other memory objects. The maximum energy vertex to be allocated to the scratchpad is then greedily selected. This vertex is removed from the conflict graph G and appended to the list L and the unallocated scratchpad size (Rem_SPSize) is reduced appropriately.

A memory object present in the scratchpad does not conflict with the memory objects present in the cache. The energy of the conflicting memory objects is thus reduced by removing the vertex and the adjacent edges from the conflict graph. The energy consumption of a memory object x_i is computed according to the energy model proposed in subsection 3.4. The heuristic iterates as long as there exists a memory object which can be placed on the scratchpad without violating the scratchpad size constraint. On termination, a list of memory objects to be allocated onto the scratchpad is returned. The time complexity of the heuristic is $O(ScratchpadSize * (|X| + |E|))$ if we precompute and store the energy consumption of each memory object x_i at the end of each "while loop" iteration.

4.2 Integer Linear Programming

In order to formulate an *Integer Linear Programming* problem, we need to linearize the scratchpad allocation problem. This can be achieved by replacing the non-linear term $l(x_i) * l(x_j)$ of equation (11) by an additional variable $L(x_i, x_j)$:

$$\begin{aligned} E(x_i) = {} & f_i * E_{SP} + \\ & f_i * [E_{Cache\_hit} - E_{SP}] * l(x_i) + \\ & [E_{Cache\_miss} - E_{Cache\_hit}] * [\sum_{j \in N_i} L(x_i, x_j) * m_{ij}] \end{aligned} \tag{15}$$

In order to prevent the linearizing variable $L(x_i, x_j)$ from taking arbitrary values, the following linearization constraints have to be added to the set of constraints:

$$l(x_i) - L(x_i, x_j) \geq 0 \tag{16}$$

$$l(x_j) - L(x_i, x_j) \geq 0 \tag{17}$$

$$l(x_i) + l(x_j) - 2 * L(x_i, x_j) \leq 1 \tag{18}$$

The objective function E_{Total} and the scratchpad size constraint remain unchanged (cf. equations (12) and (13)).

A commercial ILP Solver [6] is used to obtain an optimal subset of memory objects which minimizes the objective function. The number of vertices $|X|$ of the conflict graph G is equal to the number of memory objects, which is bounded by the number of basic blocks in the program code. The number of linearizing variables is equal to the number of edges $|E|$ in the conflict graph G. Hence, the number of variables in the ILP problem is equal to $|X| + |E|$ and is bounded by

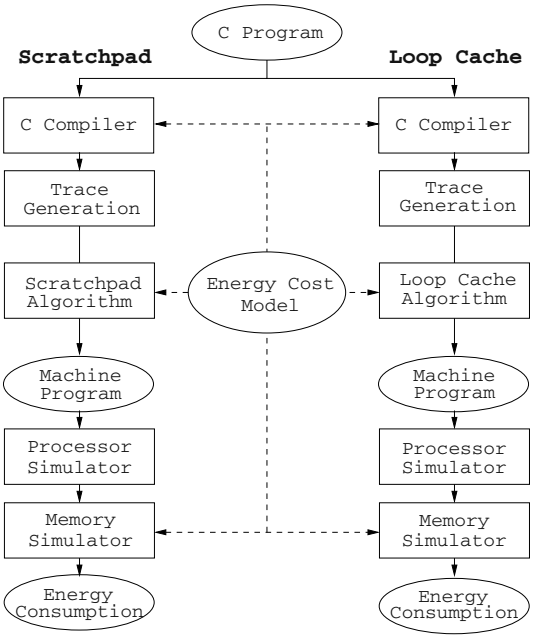


Fig. 4. Experimental Workflow

$O(|X|^2)$. The actual runtime of the used ILP solver [6] was less than one second on a Sun-Blade 100 running at 500 MHz for a conflict graph containing 455 vertices. The computation times may be expected to increase if non-commercial tools (e.g. lp_solve [5]) are used. In the next section we describe the experimental setup used for conducting experiments.

5 Experimental Setup

The experimental setup consists of an ARM7T processor core, onchip instruction and data caches, an onchip scratchpad and an off-chip main memory. The used instruction cache has a direct-mapped organization since this architecture has been found to be most suitable for low-power instruction caches [20]. The capacity of the instruction cache was selected according to the size of the corresponding benchmark. We determine the effect of allocation techniques for scratchpad on the energy consumption of the instruction memory subsystem. The *cacti* cache model [22] was used to calculate the energy consumption per access to a cache, loop cache and scratchpad memory, all assumed to be onchip and in 0.5μm technology. The loop cache was assumed to be able to hold a maximum of 4 loops. The energy consumption of the main memory was measured from our evaluation board [18].

Experiments were conducted according to the workflow presented in figure 4. In the first step, the benchmarks programs are compiled using ENCC [7], an energy aware C compiler. Trace generation [21] is a well known I-cache perfor-

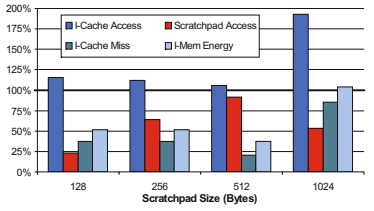


Fig. 5. Comparison of Scratchpad (Heuristic) against Scratchpad (Steinke) for MPEG

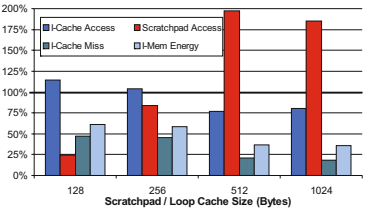


Fig. 6. Comparison of Scratchpad (Heuristic) against Loop Cache (Ross) for MPEG

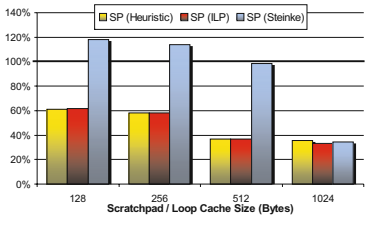


Fig. 7. Comparison of Heuristic, ILP, Steinke's and Ross's Algorithm for MPEG

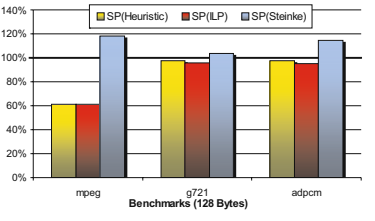


Fig. 8. Comparison of Heuristic, ILP, Steinke's and Ross's Algorithm for all benchmarks

mance optimization technique. For a fair comparison, traces are generated for all the allocation techniques. In the following step, the scratchpad allocation algorithm can either be the greedy heuristic (cf. subsection 4.1), the ILP based allocation algorithm (cf. subsection 4.2) or Steinke's scratchpad allocation algorithm [19]. The generated machine code is then fed into ARMulator [1] to obtain the instruction trace. Our custom memory hierarchy simulator [8], based on the instruction trace, memory hierarchy and the energy cost model, computes the aggregate energy consumed by the memory subsystem.

For the loop cache configuration, the loop cache is preloaded with the loops and functions selected by the allocation algorithm presented in [10]. The energy consumed by the memory subsystem is computed in a similar way, using the appropriate memory hierarchy and energy cost model.

6 Results

A subset of benchmarks from the Mediabench suite were used to substantiate our claims concerning energy savings using the proposed algorithm. The size of the scratchpad/loop cache was varied while keeping the rest of the instruction memory subsystem invariant. The number of accesses, hits and misses to every memory in the hierarchy were counted. Based on this information and the energy model (subsection 3.4), energy consumption was computed.

Table 1. Overall Energy Savings

Benchmark (size)	MemSize (bytes)	Energy Consumption (μJ)				Improvement(%)		
		SP (Heu)	SP (ILP)	SP (Steinke)	LC (Ross)	SP(Heu) vs. SP(ILP)	SP(Heu) vs. SP(Steinke)	SP (Heu) vs. LC (Ross)
adpcm	128	3567	3397	2763	2998	-5.0	-29.1	-19.0
(1 KB)	256	1744	1695	2040	1784	-2.8	14.6	2.3
	512	225	—	1400	1140	—	84.0	80.3
g721	128	7565	7393	8012	7739	-2.3	5.6	2.2
(4.7 KB)	256	6412	5984	6321	6446	-7.1	-1.4	0.5
	512	5249	4478	4469	6131	-17.2	-17.4	14.4
	1024	2566	2107	3033	6207	-21.8	15.4	58.7
mpeg	128	6318	6324	12161	10293	0.1	48.0	38.6
(21.4KB)	256	5983	5989	11697	10266	0.1	48.9	41.7
	512	3779	3755	10157	10291	-0.6	62.8	63.3
	1024	3709	3419	3579	10336	-8.5	-3.6	64.1
						-6.5	20.7	28.9

Figure 5 displays the energy consumption along with all its respective parameters (i.e. scratchpad accesses, cache accesses and cache misses) of the proposed heuristic for the MPEG benchmark. The instruction cache size was set to 2k for these experiments. All the results are shown as percentages of Steinke's algorithm [19], with the parameters of that algorithm being denoted as 100%. It is interesting to note that in spite of higher I-cache accesses and lower scratchpad accesses, the heuristic reduces energy consumption against Steinke's algorithm. The substantially lower I-cache misses are able to over-compensate for higher I-cache accesses and result in reduced energy consumption. The justification for this is that Steinke's algorithm tries to reduce energy consumption by increasing the number of accesses to the energy efficient scratchpad. In contrast, our heuristic tries to reduce I-cache misses by assigning conflicting memory objects to the scratchpad. Since I-cache misses account for a significant portion of energy consumption, the heuristic is able to conserve up to 63% energy against Steinke's algorithm. In one case (1024 bytes scratchpad), Steinke's algorithm performs marginally better than our approach. For this setup, moving (instead of copying) the memory objects seems to completely change the program's cache conflict behavior. However, there is no way of foreseeing this kind of effect when applying Steinke's algorithm and it might happen that instead of reducing the cache misses, cache performance and energy consumption are deteriorated since the algorithm doesn't account for cache behavior.

In fig. 6, we compare a scratchpad allocated with our heuristic against a loop cache preloaded with Ross's algorithm [10]. Similar to figure 5, all results are shown as percentages of the corresponding parameters of Ross's algorithm [10]. For small scratchpad/loop cache sizes (128 and 256 bytes), the number of accesses to loop cache are higher than those to scratchpad. However, as we increase the size, the loop cache's performance is restricted by the maximum number

of only 4 preloadable memory objects. The scratchpad, on the other hand, can be preloaded with any number of memory objects as long as their aggregate size is less than the scratchpad size. Moreover, the number of I-cache misses is substantially lower if a scratchpad is used instead of a loop cache. Consequently, a scratchpad is able to reduce energy consumption at an average of 52% against a loop cache for the MPEG benchmark.

In figure 7, we compare the energy consumption of different scratchpad allocation algorithms (viz. Heuristic, ILP and Steinke's) for scratchpad based systems and that of Ross's algorithm [10] for loop cache based systems. As earlier, the energy consumption due to Ross's algorithm is denoted as 100% while the energy consumption of the scratchpad allocation algorithms are denoted as percentages of Ross's algorithm. A couple of interesting points can be noted for the figure. Firstly, the heuristic performs fairly close to the optimal solution obtained by the ILP based algorithm. Secondly, for the smaller sizes (128 and 256 bytes), loop cache performs better than the scratchpad allocated with Steinke's algorithm [19], while the opposite is true for larger sizes. Figure 8 depicts the comparison of scratchpad allocation algorithms and Ross's algorithm for all benchmarks. A scratchpad and a loop cache of 128 bytes was assumed to be present in the memory hierarchy. The instruction cache size was set to 1k and 128 bytes for *g721* and *adpcm*, respectively. Observations similar to the previous figure can be noted.

Finally, table 1 summarizes the energy consumption for scratchpad and loop cache allocated with the corresponding allocation algorithms.

7 Conclusion and Future Work

In this paper, we model the cache-behavior based scratchpad allocation problem as a generic non-linear optimization problem. The problem is solved near-optimally using a heuristic and also optimally using an ILP based approach. The energy consumption of the heuristic is on average a meagre 6.5% away from that of the optimal solution. The presented techniques reduce the energy consumption of the system against a published algorithm. An average reduction of 20.7% in energy consumption due the heuristic is observed. In addition, we also demonstrate that the simple scratchpad memory allocated with the presented techniques outperforms a preloaded loop cache. Average energy savings of 28.9% are observed for the proposed heuristic and even higher values can be reported for ILP based allocation algorithm. The presented techniques can be easily extended to handle a variety of complex memory hierarchies.

References

1. ARM. *Advanced RISC Machines Ltd.* www.arm.com.
2. O. Avissar, R. Barua, and D. Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. *IEEE Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

3. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proc. of 10th International Symposium on Hardware/Software Codesign*, Colorado, USA, May 2002.
4. N. Bellas, I. Haji, C. Polychronopoulos, and G. Stamoulis. Architectural and Compiler Support for Energy Reduction in Memory Hierarchy of High Performance Microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design ISPLED*, Monterey, CA, USA, August 1999.
5. M. Berkelaar. *lp_solve: a Mixed Integer Linear Program solver.* available from: ftp://ftp.es.ele.tue.nl/pub/lp_solve.
6. CPLEX. *CPLEX Ltd.* www.cplex.com.
7. Department of Computer Science XII, University of Dortmund. *ENCC.* http://ls12-www.cs.uni-dortmund.de/research/encc.
8. Department of Computer Science XII, University of Dortmund. *MEMSIM.* http://ls12.cs.uni-dortmund.de/∼wehmeyer/LOW_POWER/memsim_doc.
9. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide To the Theory of NP-Completeness.* Freeman, New York, USA, 1979.
10. S.C.A Gordon-Ross and F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. *Computer Architecture Letters*, January 2002.
11. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3. edition, 2003.
12. M. Kamble and K. Ghosh. Analytical Energy Dissipation Models for Low Power Caches. In *Proceedings of the International Symposium on Low Power Electronics and Design ISPLED*, Monterey, CA, USA, August 1997.
13. L.H. Lee, B. Moyer, and J. Arends. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with small Tight Loops. In *Proceedings of the International Symposium on Low Power Electronics and Design ISPLED*, San Diego, CA, USA, August 1999.
14. P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig. Fast, predictable and low energy memory references through architecture-aware compilation. In *Proceedings of the Asia and South Pacific Design Automation Conference ASPDAC 2004 (to appear)*, 2004.
15. MOTOROLA. *Motorola Inc.* http://e-www.motorola.com/files/shared/doc/selector_guide/SG1001.pdf.
16. P.R. Panda, N.D. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip.* Kluwer Academic Publishers, Norwell, MA, 1999.
17. P. Pettis and C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation.* ACM SIGPLAN, June 1990.
18. S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In *Proceedings of International Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*, Yverdon-Les-Bains, Switzerland, Sep. 2001.
19. S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of Design Automation and Test in Europe DATE*, Paris France, March 2002.
20. C.-L. Su, , and A.M. Despain. Cache Design Trade-Offs and Performance Optimization: A Case Study. In *Proceedings of the International Symposium on Low Power Design ISLPD*, pages 63–68, 1995.

21. H. Tomiyama and H. Yasuura. Optimal Code Placement of Embedded Software for Instruction Caches. In *Proceedings of the 9th European Design and Test Conference ET&TC*, Paris, France, March 1996.
22. S.J.E. Wilton and N.P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31(5), May 1996.

Online Prediction of Battery Lifetime for Embedded and Mobile Devices

Ye Wen, Rich Wolski, and Chandra Krintz

Computer Science Department, University of California, Santa Barbara
{wenye, rich, ckrintz}@cs.ucsb.edu

Abstract. This paper presents a novel, history-based, statistical technique for online battery lifetime prediction. The approach first takes a one-time, full cycle, voltage measurement of a constant load, and uses it to transform the partial voltage curve of the current workload into a form with robust predictability. Based on the transformed history curve, we apply a statistical method to make a lifetime prediction. We investigate the performance of the implementation of our approach on a widely used mobile device (HP iPAQ) running Linux, and compare it to two similar battery prediction technologies: ACPI and Smart Battery. We employ twenty-two constant and variable workloads to verify the efficacy of our approach. Our results show that this approach is efficient, accurate, and able to adapt to different systems and batteries easily.

1 Introduction

Power is a critical resource for battery-powered embedded systems and mobile devices. As such, battery life must be monitored and managed within these systems to ensure maximum efficiency and effective prioritization on behalf of system users. While compile-time optimization of application code can reduce the battery consumption of individual applications, operating system support is needed to manage the combined power consumption of multiple programs executing in concert. Providing this support at the operating system level requires the ability to *predict*, accurately, remaining battery life given a dynamically changing system workload.

In this paper, we investigate an on-line statistical approach to battery lifetime prediction that combines recently observed power dissipation "history" with pre-computed off-line benchmark measurements. By dynamically incorporating on-line measurements, our approach is able to make predictions that take into account varying workloads, the "recovery effect" that batteries experiences when they are unloaded, and the charging-cycle effect that changes battery performance as batteries are repeatedly recharged.

Much of the prior work investigating battery dissipation and prediction is analytical, simulation based, or both [4, 6, 1, 11]. These systems attempt to provide accurate dissipation predictions off-line, for use in design or analytical contexts. Efficient analytical methods such as [14], and [15] consider the problems of online prediction, but do not include the statistical components needed to rapidly

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 57–72, 2004.

analyze dynamically changing workloads and operating conditions. While some approaches have considered statistical characteristics in combination with analytical models, they focus exclusively on battery dissipation in isolation [13, 16]. To be useful in an operating system resource management context, however, a battery lifetime prediction technique must be

- **fast** enough to make predictions so that real-time or near real-time decisions can be made,
- **power-efficient** enough to be run on the battery-powered device itself,
- **dynamically adaptive** so that it can take into account different user workloads, and environmental operating conditions (e.g. ambient temperature, battery recharge count, etc.), and
- **portable** so that a variety of battery and device combinations can be supported by the same operating system.

To address these challenges, our approach treats operating system power measurements from the battery as coming from a "black box." We use off-line profiling of the installed battery to establish a *reference signature* for its observed dissipation curve. We then use fast, on-line regression to predict deviations from this signature. Thus, our method uses benchmark data from the battery (in the form of a reference signature) to parameterize a statistical model that we evaluate on-line. Because the system uses measurements taken in the operating system, it is portable between devices and batteries. By using immediate on-line history, the system adapts to dynamic changes in system conditions.

We investigate the efficacy of our work by empirically evaluating our methods using the popular HP iPAQ running the Linux operating system. All of the necessary data for our method is obtained through standard hardware and operating system interfaces provided by Familiar Linux, a commonly used Linux implementation for iPAQ devices. We compare our results to those provided by two native Linux battery lifetime prediction systems: the Advanced Configuration and Power Management Interface (ACPI) and Smart Battery [5]. While considered to provide very rough estimation of battery lifetime, these utilities nonetheless meet the requirements that we describe above. That is, they implement fast, on-line, portable prediction method at the operating system level. Our method combines the attractive online features of ACPI and Smart Battery with prediction accuracy, and thus constitutes an fast, accurate, and adaptive prediction mechanism that can be used as the basis for "power-aware" operating system design.

To describe this work in greater detail, the remainder of this paper is organized as follows. In Section 2, we describe the methodology more completely. Section 3 provides an empirical evaluation of our method through direct experimentation and in Section 5 we draw brief conclusions from our investigation.

2 History-Based Battery Lifetime Prediction

Our methodology consists of three components: a *reference signature* from the battery, a *curve transformation function* that changes coordinates to make fast

prediction possible, and a *fast linear fitting* technique that makes predictions in the transformed space. The baseline observation that makes this methodology possible is that for constant but differing workloads, the "shape" of the battery dissipation curve is similar. Thus, using the trajectory produced by one workload, the lifetime implied by other constant workloads, can be predicted accurately. By transforming the coordinate space into one where simple linear fitting techniques are applicable, the predictions can then be made using computationally efficient techniques.

2.1 Linearity, Reference Curve and Voltage Curve Transformation

To determine the reference signature of a battery, we execute constant-power workloads on a quiescent system. A constant-power workload consists of repeated executions of single program instance from full battery charge until battery expiration. We describe the individual program instances in Section 3, but for the purpose of describing our methodology, the salient feature is that the power drain is constant with respect to the application workload, e.g., there is only a single program in each workload.

Linux permits application access to the voltage level reported by the battery on the iPAQ. During each complete benchmark run, the power level is recorded periodically to produce a drain trajectory. This trajectory can be expressed by function $F : t \rightarrow v$, mapping time t to battery voltage level v. v's value is between the open circuit voltage (approximately the voltage when the battery is fully charged) and the cut-off voltage (the voltage when the battery dies). In Figure 1(left), we show two typical voltage curves, which are obtained by repeatedly running benchmark programs (*IMem* – a memory read benchmark – and *IMemWC* – a cache-write benchmark – in this case) on an HP iPAQ until the battery dies. The x-axis represents the time and y-axis represents the voltage level. We describe the full experimental setup and benchmark information more completely in Section 3.

The voltage curves in the left graph of Figure 1 are inherently non-linear due to the internal electrochemical characteristics of the battery. This non-linearity limits our ability to predict remaining battery life efficiently. If we treat the trajectories as invertible continuous functions, however, we can make the observation that

$$V = F_1(t_1) = F_2(t_2) \tag{1}$$

for voltage V, and dissipation functions F_1 and F_2. If we use $\Gamma_{1,2}$ to represent the relationship between t_1 and t_2 under F_1 and F_2 for any voltage level V, we have:

$$F_1(\Gamma_{1,2}(t_2)) = F_2(t_2) \tag{2}$$

Furthermore, we can see that the voltage curves of constant workloads have very similar shapes. Based on this shape-similarity, we make the further simplifying assumption that the timing relationship $\Gamma_{i,j}$ for any two constant workloads, F_i and F_j, is a series of functions with the same form but different parameters,

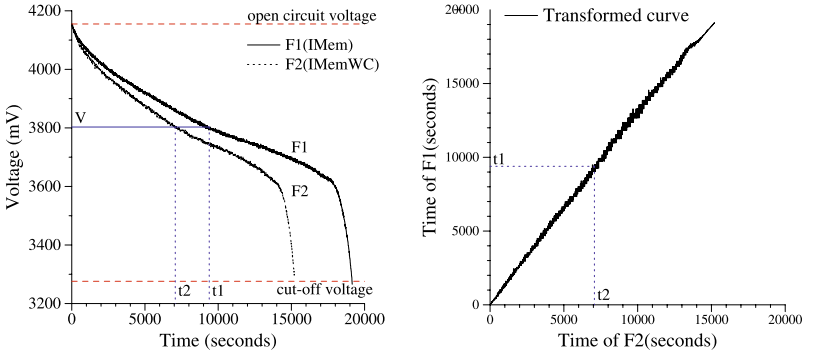


Fig. 1. Timing relationship between two voltage curves of constant workloads. F_1 is the curve of the workload that is generated by repeatedly running benchmark *IMem*. And F_2 is the curve of benchmark *IMemWC*. The left graph shows that for some voltage value V, F_1 reaches V at time t_1 and F_2 reaches V at time t_2. The right graph shows the linear relationship between t_1 and t_2 for any V

denoted as $\Gamma(\phi_{i,j}, t)$, where $\phi_{i,j}$ is a specific set of parameters for F_i and F_j. For the curves in Figure 1, we now have:

$$F_1(\Gamma(\phi_{i,j}, t_2)) = F_2(t_2) \tag{3}$$

So:

$$\Gamma(\phi_{i,j}, t_2) = F_1^{-1}(F_2(t_2)) \tag{4}$$

We plot the Γ function for curves F_1 and F_2 in the right graph of Figure 1. The x-axis is the time of F_2 and the y-axis is the time of F_1. In this graph, the Γ curve appears very close to a linear function. Our experiments show that this strong linearity actually exists between any pair of constant workloads. Figure 2(left) shows another three Γ curves for pairs of constant workloads. The axes are similar to those in the right graph of Figure 1.

If we use one specific voltage curve of constant workload as the reference, denoted as F_{ref}, the Γ function between any curve F and the reference curve F_{ref} can be expressed by:

$$\Gamma(\phi, t) = F_{ref}^{-1}(F(t)) \tag{5}$$

Since Γ can be approximated by linear function, let $\phi = (a, b)$, and we have:

$$F_{ref}^{-1}(F(t)) = a * t + b \tag{6}$$

Note that here a and b vary for different constant workloads. The Γ function actually shows not only the timing relationship between two workloads, but also indicates the size of the load: the larger the slope of the curve, the higher is the power consumption and the shorter does the battery lifetime extend. We refer to the Γ function as the *transformed voltage curve*. Using a reference voltage curve, we can transform any non-linear voltage curve of constant workload into a linear

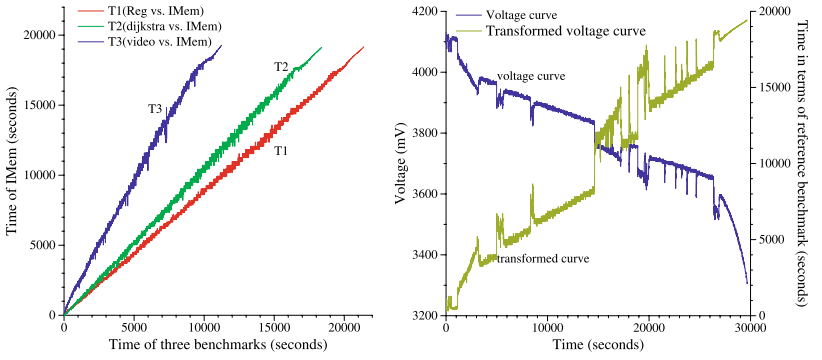


Fig. 2. Timing relationship (Γ function). The left graph plots the timeing relationship for three pairs of constant loads: *Reg* vs. *IMem*, *dijkstra* vs. *IMem*, *video* vs. *IMem*. The right graph shows both the voltage curve and the transformed curve (also based on *IMem*) for *real.load.5*

form, which is friendly to fast statistical methods, e.g., linear curve fitting, for remaining battery lifetime prediction.

In real life, workloads may not be constant. We can approximate the power consumption of a variable workload with a piecewise constant curve. Such an approximation is reasonable since the tasks of a workload are composed of consecutive execution of a sequence of operations, whose power consumption can be regarded as constant. Given this assumption, the transformed voltage curve for a variable workload should appear as a piecewise linear curve under ideal conditions.

However, the actual voltage curve of a variable load also exhibits the *recovery effect*. The *recovery effect* refers to the phenomenon that a battery regains some capacity when the load decreases. Through observation, we find that the recovery effect occurs whenever the load changes. If the load decreases, the voltage will "jump up" to a higher value instead of monotonically decreasing. If load increases, the voltage will "jump down" sharply. On the transformed curve, the "jumping" direction is inverted because an advance in time equates to a drop on voltage. Figure 2(right) shows the original voltage curve (from top-left to bottom-right) of a variable workload *real.load.5* and its transformed curve (from bottom-left to top-right) in the same graph. Both curves share the same x-axis, which represents the workload execution time. The left y-axis shows the voltage level for the voltage curve; the right y-axis shows the corresponding time for the transformed voltage curve given the reference curve. Both curves exhibit fluctuations due to the *recovery effect*.

Despite the *recovery effect* and the use of piecewise linear estimation, simple statistical methods can still achieve accurate prediction on the transformed curve for variable workloads. We present this data as part of our empirical evaluation in Section 3. In addition to the *recovery effect*, our curve transformation methodology also captures a number of other challenging battery characteristics that often limit prediction accuracy performance in other techniques [9], e.g., the *rate*

capacity effect (when the battery is discharged under different workloads, it registers different capacities) and the *cycle aging effect* (battery capacity gradually diminishes after repeatedly being discharged).

Our transformation actually is equivalent to a coordination system switch (from $(time, voltage)$ to $(time, time)$). Since the reference curve is a one-to-one function, we don't lose any information during transformation. Thus, the transformed curve keeps all of the characteristics of the battery discharge that the original voltage curve describes. Based on transformed curve, and due to the nature of the statistical methods we use, our prediction methods are insensitive to all these non-ideal phenomena and can still make an accurate prediction.

2.2 Prediction Methods

In the remainder of the paper, we refer to the transformed voltage curve as the *history curve* since it provides us with a history of battery consumption by the system up to the point at which we make a prediction of remaining battery life. In addition, to make this prediction, we considered a number of different methods. We evaluate each method using the *prediction error* of each. Assume that function P_{t_0} is the function we use to model the history curve, where t_0 is present time. Let v_e be the threshold voltage with which the battery is considered exhausted. Prediction error can be expressed by:

$$error = \left| \frac{L_p - L}{L} \right| = \left| \frac{P_{t_0}^{-1}(u_e) - L}{L} \right| \tag{7}$$

where $u_e = F_{ref}^{-1}(v_e)$, L_p is the predicted lifetime and L is the actual measured lifetime. Figure 3 demonstrates the calculation of prediction error.

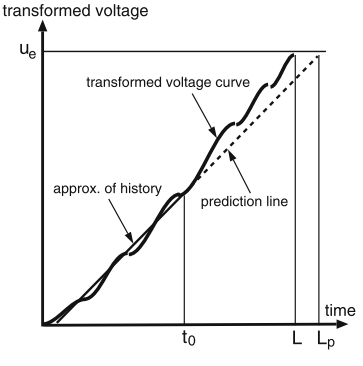


Fig. 3. Prediction error calculation

The first prediction method that we considered uses the average power consumption rate of the history curve to estimate that of the future. According to Equation (6), the slope at any point of the transformed voltage curve indicates

the magnitude of the power consumption rate (the ratio between P and P_{ref}). As such, we can model future power consumption at time t as $P_{t_0}(t) = k_{t_0}t$, where k_{t_0} is the mean slope of the history curve before t_0. We refer to this method as *Mean Slope Prediction(MSP)*. We then improved this method by making a prediction line that begins at the current point $(t_0, G(t_0))$ instead of $(0, 0)$: $P_{t_0}(t) = k_{t_0}t + G(t_0) - k_{t_0}t_0$. We call it *Mean Slope Point Prediction(MSPP)*.

We next considered a model that uses linear *Least Square Fit(LSF)*[12]. Assume that k_{lsf} and b_{lsf} are the slope and intercept of the linear regression, the prediction function will be $P_{t_0}(t) = k_{lsf}t + b_{lsf}$. We call this method *LSF Prediction(LSFP)*. As we did for Mean Slope Prediction, we also consider the efficacy of using LSFP when the prediction line starts at the current time (as opposed to the beginning of time $(0, 0)$). We call this method *LSF Point Prediction(LSFPP)*, and implement it as $P_{t_0}(t) = k_{lsf}t + G(t_0) - k_{lsf}t_0$.

All four of these methods are computationally efficient. For example, using method LSFPP or LSFP, a single prediction for a median-sized voltage curve (about $4,000$ readings) takes 250 milliseconds on average on the 206MHz StrongARM processor of the HP iPAQ; this is equivalent to about 0.25 joule of energy consumption. Since MSPP and MSP have lower computational complexity and they can be computed incrementally, they consume even less energy.

Given the history curve and prediction functions, the final question that we must address concerns the length of history that is required to make the best prediction. In Section 3.3, for each prediction method, we empirically evaluate a range of different history sizes to answer this question. This study also gives us insight into how well our techniques perform given a small amount of history.

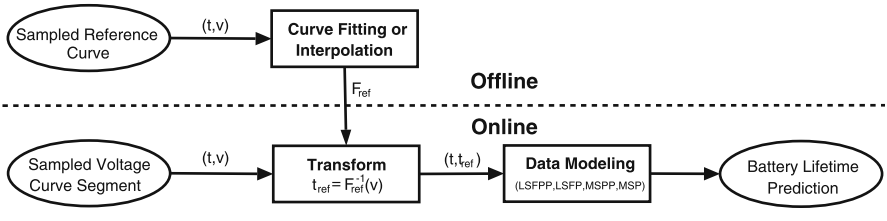


Fig. 4. Prediction procedure

Figure 4 summarizes the procedure of making a prediction. The current, sampled, voltage curve, which is a sequence of (t, v) pairs (t is a timestamp, v is voltage), is transformed using the reference curve. Next, the above prediction methods are applied to the transformed curve, which is a sequence of (t, t_{ref}) pairs ($t_{ref} = F_{ref}^{-1}(v)$), and a prediction is made. Note that the reference curve must also be transformed from a sequence of discrete pairs to a continuous mathematical form using curve fitting or interpolation. Note also that the online part of the prediction procedure includes the computation of the inverse of *reference curve* function. We use the Newton-Raphson method [12] to make the

approximation. This method is also very efficient practically. In our experiment, it takes about 3 iterations on average to get a result with an error within 0.001.

3 Evaluation

We evaluated our prediction methods using two models of HP iPAQ: H3650 and H3835. H3650 is equipped with a 1000 mAh Danionics DLP 305590 lithium-ion polymer battery and H3835 has a 1400 mAh Danionics DLP 345794 battery [3]. Since the results for these two models are similar, we only present the H3835 results in this paper; the trends, however, are the same.

We installed Familiar Linux v0.6.1 [10] and the Opie environment [10] on the iPAQ to perform all experiments. Opie provides the graphical user interface and a set of applications such as games, a media player, and a calendar, that we use as benchmarks. The iPAQ has an internal voltage sensor reporting accurate battery voltage measurements via Linux "/proc" system. We implemented a logging program that reads the current battery voltage from "/proc/asic/battery"(ACPI-like battery status report) into a file stored locally. For each benchmark, we first fully charge the iPAQ battery. We then start the workload benchmark and the logging program simultaneously. The logging program runs periodically with an interval of 6 seconds. We use this empirically selected interval since it is short enough to catch significant changes in voltage and long enough to reduce interruption. The benchmark runs continuously until the battery dies.

3.1 Workload Benchmarks and Reference Benchmark

We generated the constant workload by repeatedly running a single benchmark program, for which one-time execution time is very short (within 4 minutes). We evaluated 14 such programs. They include the benchmarks that we hand-coded to execute of a single type of instruction (*Reg* (register instructions only), *IMem*(loads, out of cache), *IMemC*(loads, in cache), *IMemW* (stores, out of cache) and *IMemWC* (stores, in cache)). In addition, we included *dijkstra*, *fft*, *ispell*, *jpeg*, *sha* and *susan*, from the MiBench Suite [7], and three multimedia programs: *audio*, *video* and *videoaudio*. Each of the MiBench programs represents one of six application categories: network, telecommunication, office, consumer, security and automotive, covering a broad range of typical embedded system applications. The three multimedia programs play MPEG format audio, video and video with audio respectively. All of these programs exercise many hardware functions in an embedded or mobile device, e.g., CPU, memory, flash, audio/video components, and backlight.

We generate part of the variable workloads by simulating the real usage of a PDA. The simulation program is composed of a set of hand-held device applications (e.g. multimedia, note-taking, etc) provided by the Opie toolkit [10]. Each application runs a specified period of time during the simulation. Different patterns of variable workloads are generated by different configurations of the simulation program. The *simu.random* workload is generated by randomly executing one of these applications in uniform distribution. The other three

Table 1. Workload description. *IMem* is used to generate *reference curve* exclusively

Benchmarks	Type	Comments
Reg	constant: single instruction	register instruction ONLY
IMem	constant: single instruction	memory read instruction, 100% cache miss
IMemC	constant: single instruction	memory read instruction, 100% cache hit
IMemW	constant: single instruction	memory write instruction, 100% cache miss
IMemWC	constant: single Instruction	memory write instruction, 100% cache hit
dijkstra	constant: single operation	shortest path algorithm benchmark
fft	constant: single operation	Fast Fourier Transform benchmark
ispell	constant: single operation	a fast spelling check benchmark
jpeg	constant: single operation	JPEG encoder/decoder benchmark
sha	constant: single operation	SHA secure hashing algorithm benchmark
susan	constant: single operation	image recognition benchmark
audio	constant: single operation	play a 210-second MP3 audio file with audio output, back light off
video	constant: single operation	play a 142-second MPEG1 video file without audio, back light on
videoaudio	constant: single operation	play a 142-second MPEG1 video file with audio output, back light on
simu.random	variable	simulated random workload, no sleep time
simu.30	variable	simulated random workload, 30% probability to sleep
simu.50	variable	simulated random workload, 50% probability to sleep
simu.70	variable	simulated random workload, 70% probability to sleep
real.load.1	variable	5 real workloads, voltage curve recorded
...		when PDA is used by people
real.load.5		

workloads (*simu.30*, *simu.50* and *simu.70*) are generated in the following way. The simulation program continuously allocates time slots of random length to either an idle mode or a specific set of applications that are specialized in similar functions (e.g. audio/video), according to a predefined distribution. During each non-idle time slot, the applications within the specific set are also executed randomly following a predetermined distribution. In this way, *simu.30* keeps the device busy during 70% of the time on average. Similarly, *simu.50* and *simu.70* have a device usage frequency of 50% and 30% respectively.

We also obtained 5 real variable workloads, whose voltage curves are recorded when users played with the iPAQ in a common way, e.g., playing games, viewing pictures and videos, listening to music, and making a schedule using the calendar. These workloads were obtained by loaning the iPAQ to individual students and then recording the power dissipation each student induced. Table 1 summarizes the total 23 constant and variable workloads.

Finally, we pick the *IMem* benchmark to generate the *reference curve*. A key contribution of our method is that any constant-workload benchmark can be used to generate *reference curve*; the fluctuations in a variable workload require smoothing if it is to be used as the *reference curve*. The reason for this is that our method relies on similarities in the *shape* of the curves; all constant-workload benchmarks exhibit similar shape, as such, they can be used as the reference curve with statistically similar results.

Since the sampled *reference curve* is composed of a sequence of discrete pairs $(time, voltage)$, we cannot use it to compute the transformed voltage, $F_{ref}^{-1}(v)$,

since it is not expressed in terms of v. Instead, we model the reference curve off-line using a high order polynomial and polynomial least square fit [12]. The *IMem* curve can be fit by a polynomial of order 15 (the coefficient of determination of the fit, R^2, is 0.99955). We then use this polynomial as the *reference curve* function on-line to make each prediction.

3.2 Results of Prediction for Constant Workloads

Given the voltage curve of a constant load, we first generated the history curve (transformed curve) using the reference polynomial. Then, for every 50th point (approximately 5 minutes), we apply each of our prediction methods, LSFPP, LSFP, MSPP, MSP, to make a prediction. We next calculate the error for each prediction point using Equation 7. Finally, we have a sequence of prediction errors ("moving errors") for the entire battery lifetime.

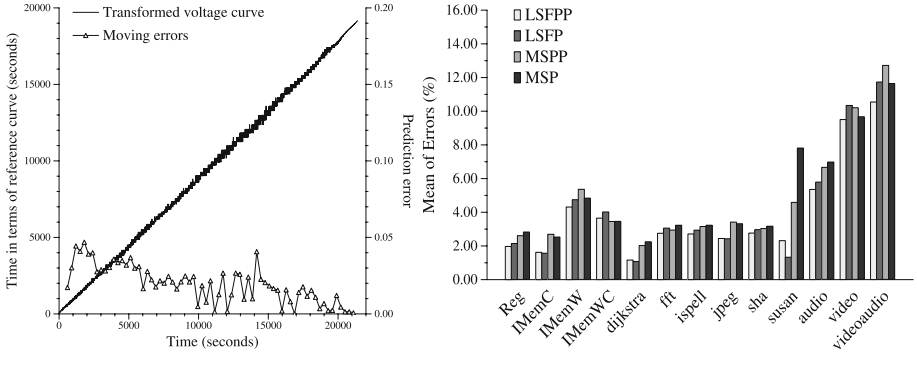


Fig. 5. Prediction performance for constant loads. The left graph shows the transformed voltage curve and moving errors for *Reg*. The right graph shows the average prediction errors for all constant loads and all methods

Figure 5(left) shows the history curve (solid line curve) and the corresponding "moving errors" (marked-line curve) for the *Reg* benchmark. Both curves share the same x-axis, which represents time. The transformed voltage curve uses the left y-axis that shows the transformed voltage in terms of reference curve time. The error curve uses the right y-axis that shows the error value. As we can see from the graph, all prediction errors are within 5%.

In Figure 5(right), we show the average prediction error for each benchmark and each method. The y-axis is average percentage error. Except for the three media benchmarks (audio, video, and videoaudio), the predictions for the constant workloads have an average error below 5% for all methods. The predictions for the media benchmarks have average errors around 10% due to the local fluctuations of power consumption when they run. Among the four prediction methods, LSF-based methods perform a slightly better than MS-based methods. This is because *least square fit* provides a better model for linear data than mean

slope does. LSFPP is the best method among all. Overall, the performance of all the four methods is similar.

3.3 Results of Prediction for Variable Workloads

We follow the same procedure to make predictions for variable workloads. A typical voltage curve and transformed voltage curve for variable workload is shown in Figure 2(right).

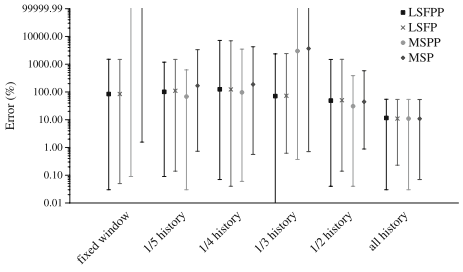


Fig. 6. Prediction performance for different history size. This figure shows the mean, max and minimum of "moving errors" of the 4 methods for *real.load.5* benchmark, using different history size

First, we explore how much history we need to make the best prediction. We tried the four methods using different history lengths: a history window with last 50 sample points, the last 1/5 , 1/4, 1/3, 1/2 of history, and all of the history. Figure 6 shows the mean (shown as the markers), maximum (shown as the top of the bar) and minimum (shown as the bottom of the bar) of prediction errors for *real.load.5* benchmark. The y-axis shows the value of prediction errors (percentage) using a log scale. The data reveals that the prediction based on the entire history has both the smallest average error and error range for most methods. We found similar results for all other benchmarks. This tells us that the methods based on recent history are not able to make an accurate prediction of the future. In the results that follow, we only use the entire history to make a prediction.

We next compare the performance of our methods against that of ACPI, a standard, commonly used, power management service that provides battery life estimation [2], and Smart Battery's rolling average algorithm [5]. The Familiar Linux reports detailed battery status through "/proc/asic/battery" file. We extract the ACPI-like battery life estimation directly from the file. We also simulate the Smart Battery's rolling average algorithm using the battery information from the file: at each prediction point, first calculate the average current within last 1 minute; then take the division of present remaining battery capacity and the average current as the present battery life estimation [5]. In Table 2, we show the mean (column 2 and 4) and standard deviation (column 3 and 5) of prediction errors using ACPI's battery life estimation, Smart Battery's rolling average algorithm and our four methods using the entire history. Columns 2 and 3 show

Table 2. Prediction performance of ACPI, Smart Battery (the rolling average) and our methods based on entire history

	real.load.5		IMemWC	
Methods	mean %	stdev %	mean %	stdev %
ACPI	60.28	27.70	42.96	71.30
rolling average	60.16	28.36	40.55	71.39
LSFPP	21.29	65.63	3.65	1.67
LSFP	20.76	66.19	4.01	1.22
MSPP	14.19	19.74	3.44	1.96
MSP	14.16	19.85	3.44	1.99

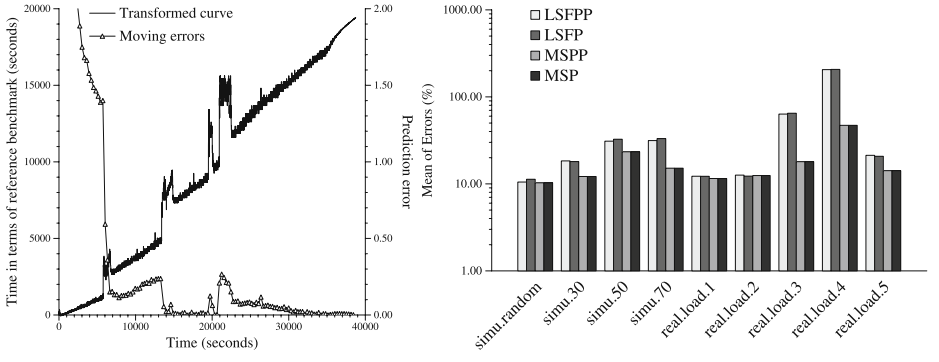


Fig. 7. Prediction performance for variable workloads. The left graph shows the transformed curve and moving errors for *real.load.4*. The right one shows the average prediction errors for all variable loads and all methods. y-axis is in log scale

data for *real.load.5*, variable workload, benchmark; columns 4 and 5 show data for *IMemWC*, constant workload, benchmark. Results for all other benchmarks are similar; each of our four methods significantly outperforms ACPI and Smart Battery's rolling average for both constant and variable workloads.

Figure 7(right) shows the average prediction errors for all of the variable workload benchmarks. At the first glance, it seems that LSF-based methods perform very poorly for variable workloads. For example, LSFPP has an average error of 205.68% for *real.load.4*.

A detailed analysis shows that the prediction by LSFPP has some huge spikes in prediction error at the start time of the experiments when there is little history available. Figure 7(left) illustrates the relationship between the transformed voltage curve and the "moving errors". The axes have are similar to those in Figure 5(left). Before time 6000, there is no program running and the power consumption is very low creating a line segment with small slope at the beginning of the transformed voltage curve. The forecaster only knows about history and it makes a prediction that the battery will last for a much longer time than actual will. Immediately after some process starts to run, the curve goes up and the forecaster begins to realize the actual power consumption. As such, the prediction error also starts to drop. During the remaining time, the prediction error is much smaller.

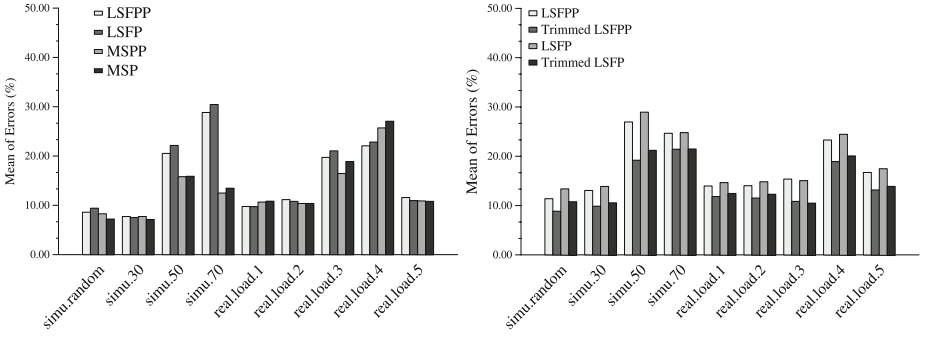


Fig. 8. Average prediction errors for variable workloads using a trim of 5%(left) and inverted curve(right)

To isolate this initial noise in the "moving errors", we calculate the mean for the last 95% of prediction errors (we call it the *trimmed prediction error*). That is, we discard the initial 5% (in terms of time) of the prediction errors to allow each forecaster to calibrate. Figure 8(left) shows the mean of the trimmed prediction errors for all variable workloads; the errors are smaller than without trimming. For example, the average prediction error of *real.load.4* by LSFPP drops to 22.08%. These results indicate that for variable workloads, MSPP and MSP methods outperform LSFPP and LSFP.

Since the least square fit is not symmetric for the x and y axes (it tries to minimize the sum of distance squared along the y-axis), we also investigated the efficacy of our methods on the curves with switched x and y axes. We call the new curves "inverted transformed voltage curve". Since it is meaningless to average the inverse of a slope that represents the power consumption, we do not apply MSPP and MSP methods to the inverted curve. We show the results in Figure 8(right) as average prediction error using variable workloads for both trimmed and non-trimmed LSF methods using the inverted transformed voltage curve. The results indicate that the prediction performance using the inverted curve is more stable. In addition, it does not suffer from the early-stage spikes in the "moving errors"; as such, it is an alternative to trimming. In general, LSFPP outperforms LSFP.

In summary, we find that LSF-based methods are slightly better than MS-based methods for constant workloads. For variable workloads, MSPP performs best. In general, we believe MSPP is the best method for our online battery lifetime prediction. MSPP is cheaper to compute and even though it is slightly less accurate under constant conditions, it is less sensitive to variability in the measurement history.

4 Related Work

Battery life estimation, an integral component of power management systems, is provided in many mobile devices via both hardware and operating system

support, such as that specified by Advanced Power Management (APM) [8] and more recently by Advanced Configuration and Power Interface (ACPI) [2]. In particular, ACPI, to which we compare our results, uses the division of remaining battery capacity and present rate of battery drain to estimate remaining battery life [2]. Smart battery is another similar technology that estimates battery life given a user-specified rate (e.g. present rate) or rolling average over a fixed interval (normally 1 minute) [5]. Such simple approaches to prediction consider only a very short discharge history and thus can be highly inaccurate, as we show in 3.3.

Another area of related research is model-based battery life estimation. Given the discharge profile of the entire lifetime of the battery, a well-designed battery model can give highly accurate battery life estimation. One such accurate model, described in [4], uses the low-level electrochemical phenomenon of battery discharge. This model is commonly used as a simulator to verify other battery models. More efficient simulation models include PSPICE-based models [6], discrete-time VHDL models [1] and a Markov chain model [11]. In [14] and [15] two efficient analytical models are proposed. The model described in [14], builds a relationship between current profile and battery lifetime. In [15], a fast prediction model is used to estimate the remaining battery capacity, which takes into account the recharge cycle aging and temperature. An approach for combining analytical models and statistical methods is proposed by [13].

These models are different from our method in that they require the complete discharge profile during a battery's life to make estimation. In other words, they make a "calculation" instead of a "prediction". Our method predicts battery life without knowledge of future workload. Model-based methods have other limitations to make them unsuitable for online, dynamic and adaptive battery life prediction, such as the need for large number of parameters and high computation cost (especially for simulation models).

In the work most related to ours [16], the authors estimate battery lifetime by exploiting the linear relationship between the system load and the drain time required to reach a specified voltage. They then apply statistical methods to make a prediction. This work differs from our work in two ways. First, the linearity they exploit is between the load and the time at which a certain voltage level is reached. As such, they must obtain a large number of load-versus-lifetime samples to generate an accurate curve fit. Using our method, only one reference voltage curve (generated off-line) is required. Second, this prior work studies only the lifetime estimation for constant workloads. It is not clear whether it can be applied to variable workloads. Our method naturally extends to variable workloads and we empirically evaluate it using both workload types.

5 Conclusions

We investigate battery lifetime prediction using a purely statistical method and only data that is readily available from the OS /proc file system. By using a statistical technique, our approach takes into account variations in workload,

application profile, and battery charge rates, particularly those caused by the recovery of the battery during idle periods. We describe a coordinate transformation that converts a dynamic voltage curve into a form that enables more robust prediction of future behavior. We implement and empirically evaluate two variations of statistical methods on the transformed curve to make predictions. The experimental results show that we are able to achieve high prediction accuracy under both constant and variable workloads.

Our approach is simple, efficient, accurate, and flexible. In addition, it can be easily incorporated into current operating systems on popular hardware. As part of future work, we plan to investigate combinations of different prediction methods to further improve the accuracy of our method. Since there is not a universal method that is best for all cases, we are also seeking a way to leverage the power of different methods.

References

1. L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. A discrete-time battery model for high-level power estimation. *In Proceedings of Design, Automation and Test in Europe*, 2000.
2. Compaq, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification, 2002.
3. Danionics lithium-ion polymer battery. `http://www.danionics.com/sw828.asp`.
4. M. Doyle, T. F. Fuller, and J. Newman. Modeling of galvanostatic charge and discharge of the lithium/polymer/insertion cell. *Journal of Electrochem Society*, 141(1):1–9, January 1994.
5. Smart Battery System Implementers Forum. Smart battery data specification(v1.1), 1998.
6. S. Gold. A PSPICE macromodel for lithium-ion batteries. *In Proceedings of Annual Battery Conference on Applications and Advances*, pages 215–222, 1997.
7. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001. Austin, TX.
8. Intel and Microsoft. Advanced power management(apm) bios interface specification, 1996.
9. D. Linden and T. B. Reddy. *Handbook of Batteries(3rd edition)*. McGraw-Hill, 2002.
10. Linux for handheld devices. `http://www.handhelds.org`.
11. D. Panigrahi, C. Chiasserini, S. Dey, R. Rao, A. Raghunathan, and K. Lahiri. Battery life estimation of mobile embedded systems. *The 14th IEEE International Conference on VLSI Design*, 2001.
12. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (Second Edition)*. Cambridge University Press, 2002.
13. D. Rakhmatov and S. Vrudhula. Time-to-failure estimation for batteries in portable electronic systems. *In Proceedings of the International Symposium on Low Power Electronics and Design*, August 2001.

14. D. Rakhmatov, S. Vrudhula, and D. A. Wallach. Battery lifetime prediction for energy-aware computing. *In Proceedings of the International Symposium on Low Power Electronics and Design*, August 2002.
15. P. Rong and M. Pedram. Remaining battery capacity prediction for lithium-ion batteries. *Conference of Design Automation and Test in Europe*, March 2003.
16. K. C. Syracuse and W. Clark. A statistical approach to domain performance modeling for oxyhalide primary lithium batteries. *In Proceedings of Annual Battery Conference on Applications and Advances*, January 1997.

Synchroscalar: Initial Lessons in Power-Aware Design of a Tile-Based Embedded Architecture

John Oliver[1], Ravishankar Rao[1], Paul Sultana[1], Jedidiah Crandall[1], Erik Czernikowski[1], Leslie W. Jones IV[2], Dean Copsey[1], Diana Keen[2], Venkatesh Akella[1], and Frederic T. Chong[1]

[1] University of California at Davis
[2] California Polytechnic State University, San Luis Obispo

Abstract. Embedded devices have hard performance targets and severe power and area constraints that depart significantly from our design intuitions derived from general-purpose microprocessor design. This paper describes our initial experiences in designing Synchroscalar, a tile-based embedded architecture targeted for multi-rate signal processing applications.

We present a preliminary design of the Synchroscalar architecture and some design space exploration in the context of important signal processing kernels. In particular, we find that synchronous design and substantial global interconnect are desirable in the low-frequency, low-power domain. This global interconnect enables parallelization and reduces processor idle time, which are critical to energy efficient implementations of high bandwidth signal processing. Furthermore, statically-scheduled communication and SIMD computation keep control overheads low and energy efficiency high.

Keywords: Low Power Processor, 802.11(a), Programmable DSP Processor, tiled-based architectures, embedded processors.

1 Introduction

Next-generation embedded applications demand high throughput with low power consumption. Current approaches often use Application-Specific Integrated Circuits (ASICs) to satisfy these constraints. However, rapidly evolving application protocols, multi-protocol embedded devices, and increasing chip NRE costs all argue for a more flexible solution. In other words, we want the flexibility of a programmable DSP with energy efficiency more similar to an ASIC. We propose the Synchroscalar architecture, a tile-based DSP designed to efficiently meet the throughput targets of applications with multi-rate computational subcomponents.

In designing Synchroscalar, we focused on three key features of ASICs that lead to their energy efficiency – high parallelism, custom interconnect, and low control overhead. Parallelism is important in that it allows the frequency of an

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 73–85, 2004.

architecture to be reduced linearly with investment in logic, modulo communication. This linear reduction, when coupled with voltage scaling, yields a quadratic decrease in power and a linear decrease in system energy. Low communication latency, however, is important in maintaining the parallelism necessary for these energy gains. ASICs accomplish low latency through custom interconnect. We find that, in the low frequency domain, a tile-based processing architecture can use segmentable global busses to achieve low latency with high energy efficiency. Control overhead of the busses is kept low by using statically scheduled segmentation and data motion. Control overhead of the tiles can be reduced by grouping columns into SIMD execution units.

In the remainder of this paper, we provide an overview of the Synchroscalar architecture to establish the context of our study. Then we provide some simple tile and interconnect models which we used to guide our design. We use these models to conduct an analysis of FIR, FFT, Viterbi, and AES kernels running on different points in the design space. We discuss our intuitions from this analysis and conclude with future work for our project.

2 Synchroscalar Architecture

In this section, we introduce the proposed Synchroscalar architecture and the rationale behind it. As noted in the previous section, we were motivated by the need for an embedded architecture with the flexibility of a general purpose processor (DSP) and the power efficiency of an application specific integrated circuit. We examined ASIC implementations of Viterbi, FFT, AES, FIR and found that the key sources of the power efficiency of an ASIC are

- Parallelism, multiple clock and voltage domains
- Customized interconnect mirroring the dataflow inherent in the computation
- Distributed memory to provide high bandwidth
- Customized functional blocks to implement the computation
- Absence of instructions, removing instruction cache accesses and decode logic

If we want to approach the efficiency of an ASIC, our architecture should retain as many of the key strengths of an ASIC as possible. This directs us towards a tiled-based multiprocessor architecture with multiple clock and voltage domains, reconfigurable interconnect, and low-overhead SIMD control.

Abstractly, Synchroscalar is a two dimensional array of processing elements (PEs), each column potentially operating at different fixed frequencies and hence voltage. There is a single vertical bus connecting the elements in a column, and these vertical buses are connected by a single horizontal bus for communication between columns. In reality, in order to reduce the distance between PEs in a single column, the column is folded over. There are PEs on both sides of the vertical bus. That is the basis for Synchroscalar, as shown in Figure1 (we do not plan to support dynamic frequency/voltage scaling at present). Because of the data-parallel nature of computation, each PE can be viewed as one functional unit of a SIMD machine. There is a SIMD controller for each pair of columns.

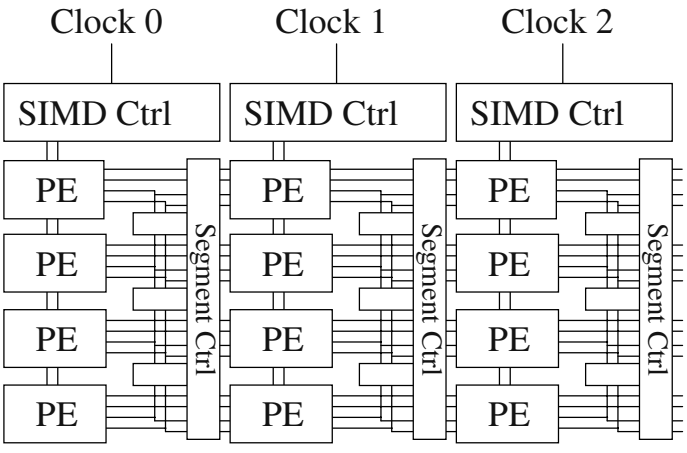


Fig. 1. The Synchroscalar Architecture

Each PE (tile) has a single DSP engine with two functional units, SRAM, register file, and communication interfaces. For brevity, we will refer to this cluster of bus, two columns, and SIMD controller as a single column. Although the tiles are SIMD, the communication patterns are not identical, so programmable engines are required for controlling communication.

2.1 Programming Model

The architecture of Synchroscalar is motivated by Synchronous Dataflow (SDF) model of computation [2, 3, 4]. DSP design environment tools created by Synopsys and Cadence use this model.

SDF is a subset of general purpose dataflow that restricts the number of data values produced and consumed by an actor to be a constant. The restriction imposed by the SDF model offers the advantage of static scheduling and decidability of key verification problems such as bounded memory requirements and deadlock avoidance [8] Synchroscalar can be viewed as a architecture to support SDF computation model efficiently. This predictability is crucial to providing the generality of programming units while retaining much of the efficiency of ASICs.

2.2 Clock and Voltage Domains

Clock and voltage domains are per-column, with the task parallelized within the column. Tasks can be mapped to different columns depending on their computational requirements. This mapping is crucial to performance, because once set, the voltage and frequency of a column may not change. Mapping algorithms must be developed to provide minimize communication and maximize power savings. Computationally-intensive tasks are performed at the best available

frequency and voltage that meets the performance requirements. Other tasks can be mapped to columns operated at lower frequency and voltage.

We employ *rational clocking*[15] for the frequencies of different columns. If f_m and f_n are the frequencies of two columns of PEs then $f_m/f_n = M/N$ where M and N are integers. While this allows a wide range of selection of frequencies, the relation between the two frequencies provides the predictable communication points between the domains required for statically scheduled communication. Rational clocking eliminates the synchronization overhead with asynchronous or GALS systems while still giving us the flexibility of different frequency domains.

ASICs benefit from high-bandwidth, low-latency communication provided by custom interconnects. We exploit low clock frequencies and static scheduling to maximize throughput while minimizing latency. Static scheduling is required to maintain guaranteed performance. Although the clock frequencies are low enough to traverse a column in a single cycle, we segment the bus in order to increase the usable bandwidth. Segment controllers are turned on or off by signals from a central per-column segment controller. As shown in Fig.1, the bus connecting two columns of PEs is partitioned into segments [23] by segment controllers.

The column segment controllers are small state machines which can be reconfigured for each algorithm. By suitably controlling the segment controllers, the bus can perform several parallel communications. For instance, if all the controllers are turned off, the bus becomes a broadcast bus, all PEs able to receive the same data. Alternatively, two messages can pass between neighboring columns using the same wires in different segments if the segment controller between them is on. The tasks are mapped to the tile architecture such that the communication between the PEs is minimized. Highly communicating tasks are assigned to neighboring PEs. This reduces the number of segments the data has to travel, and hence saves power.

2.3 SIMD Control

In order to reduce the cost of instruction fetch and decode, a single SIMD controller sends instructions to the PEs in a column. The SIMD controller performs all control instructions, only forwarding computation instructions to the PEs. To communicate data (used for conditional branches), the SIMD controller is connected to the segmented bus with the PEs.

In order to use branch prediction, there needs to be a mechanism to squash instructions that have already been sent to the processing elements. Instead, we provide a short pipeline in the control unit to calculate branches quickly, and delay instructions from reaching the processing elements. This introduces a single-cycle stall for each conditional branch. For zero-overhead loops, there is still no delay, because the PC is used for decision-making, not the actual instruction. Our implementation incurs no extra overhead for these loops which are critical to DSP performance.

3 Framework

With the Synchroscalar architecture and motivation for context, we now present a general framework within which to evaluate the surrounding design space. The framework will use some simple first-order models of tile and interconnect power, validated with datapoints in the literature and VHDL designs of custom Synchroscalar elements. Although our models are by necessity abstract enough to cover the design space, we argue that the important scaling effects are captured and that our qualitative conclusions are valid.

3.1 Tile Model

We use the voltage frequency scaling given by the Newton's alpha law f= k*$\frac{(V_{dd}-V_t)^\alpha}{Vdd}$ [14]. This equation gives the voltage-frequency scaling for a given technology. We have modeled a ring oscillator in SPICE using the Berkeley Predictive Technology Model (http://www-device.eecs.berkeley.edu/ ptm/) to get a better feel for the acceptable range of supply voltage and threshold voltage. This enables us to project the models into 90 nm and 45 nm technology.

Our tile is based on the low power 16-bit VLIW DSPs similar to the Intel-ADI MSA-based Blackfin[7] and the SPXK5 from NEC [19]. The minimum core power is assumed to be 0.07mW/MHz similar to [19]. (We are in the process of finishing a detailed VHDL model for the tile and and validating this assumption). The SRAM power is given 0.02mA/MHz for 32kB of memory. This number was obtained from the circuit given in [12], by scaling for technology and size.

3.2 Interconnect Model

The interconnect model is largely based on the data given in [6]. We find that the gate and drain capacitances are orders of magnitude smaller than the wire capacitance per unit length. We thus model only the wire capacitance. The drain-source capacitance of the segmenters and the gate and drain capacitances of the drivers are ignored. In 0.18u tech, the gate capacitance of a minimum sized transistor is about 1-2fF [6]. This value is expected to remain constant over shrinking process technologies. The projected value, in 0.13u tech, of wire capacitance of a semi-global wire, per unit length is 387fF/mm. The chip length is about 10mm and hence the wire capacitance is about 3870fF. This suggests that even if the drivers and repeaters are 10-times the minimum size, their capacitance is about 20fF. If there are 8 drivers for each bus, it adds only 160fF to the wire capacitance.

We are in the process of completing VHDL models for the segment controllers, SIMD controller and the communication interfaces. We plan to augment our results with this in the future, but we believe that they are unlikely to change the major trends in the results reported here.

4 Applications

The main objective of this paper is an exploration of the design space defined by the goals of the Synchroscalar architecture. Specifically, we are interested in the impact of various architectural parameters such as the number of tiles, the interconnect structure, the width of the buses on the power while meeting the performance constraints of an application.

For an initial driving application, we choose the 54 Mbps 802.11(a) wireless LAN physical layer. This is currently outside the scope of DSP processors and is currently done with ASICs or DSPs with co-processors for the computationally intensive applications. The computationally challenging aspects of 802.11(a) are Viterbi decoder, FFT, and large FIR filters for equalization. We will evaluate each of these function on the Synchroscalar architecture. We derive the performance (throughput) targets for each function so that we can meet the 54 Mbps data rate. In addition we also use the Advanced Encryption Standard (AES) as a benchmark as it contains very different kind of computation, intensive on bit manipulation and table look-ups, to see how our architecture fares on such workloads.

The FIR filter is used in the equalization function in the OFDM receiver. We model a 128-tap FIR filter and assume that the data rate is 64 Mbps. We also model a 128 point FFT and assume the data rate is 256 Mbps. FFT and IFFT are key components of the OFDM receiver. For the Viterbi Decoderwe assume the constraint length for the decoder K=7 and the data rate is 54 Mbps. This is the most computation intensive part of the OFDM receiver.

Our initial experimental procedure is as follows:

1. Write the function in C and verify using Blackfin Visual DSP simulation environment
2. Replace the performance critical sections of the code with Blackfin assembly code, to achieve optimal performance. This corresponds to the implementation on a single tile.
3. Next map the application into multiple tiles and using a homebrewed tool to assist in pruning the search space.
4. Manually insert the communication instructions
5. Estimate the clock cycle count for the application.
6. Using the power model for the interconnect and the tile described in the previous section, estimate the power. The parameterized power models were described in Excel and that was used to generate the graphs reported in the next section.

While an extensive cycle-level simulation infrastructure is currently under development, we felt that hand-counts were appropriate for guiding the early design of the architecture. In particular, our driving signal processing applications are very amenable to hand-analysis as their computations are focused on a small number of kernels.

5 Results

Our results focus on several key design questions. We explore the parallelism available in each algorithm by varying the number of processing tiles, the communication bandwidth necessary through varying global bus widths, and the power efficiency of communication by exploring segmented buses.

5.1 Architectural Configurations

Figure 2 shows that as the number of tiles increases, there is the traditional tradeoff between computation and communication, but performance is not our goal. As the performance increases, we lower the clock frequency to maintain

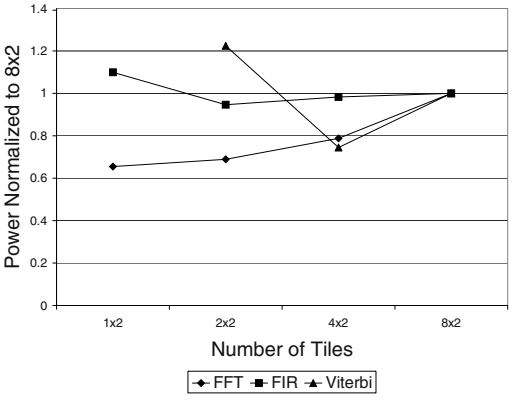


Fig. 2. Power required as the number of tiles increases

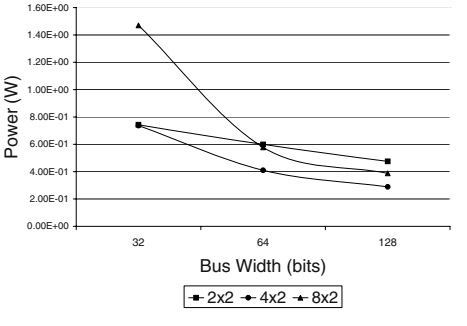


Fig. 3. Viterbi Decoder power as bus width increases for various tile configurations

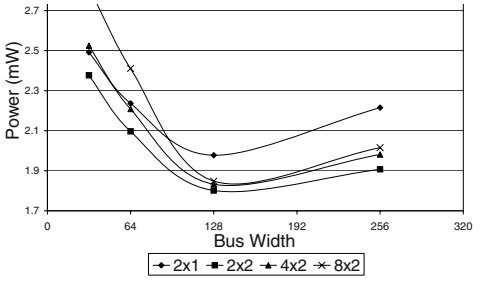


Fig. 4. FIR Filter power as the bus width increases for various tile configurations

a constant performance target, allowing a decrease in voltage. Note that this is not done dynamically. Each experiment with a different number of tiles is a completely different instance of the program. So, for each instance, we provide the lowest frequency / voltage to maintain the same performance. The tradeoff is then between adding processors, providing a constant increase in-power consumption, and reducing the voltage, providing a quadratic decrease in performance.

All three applications observe an initial decrease in total power, but by the 8x2 tile configuration, the decreasing returns of parallelization is outweighing the benefits voltage scaling. Thus we should provide either 2x2 or 4x2 tiles in each column.

5.2 Impact of Bus Width

We then vary bus width. Data dependencies prevent effective overlap of communication and computation. This makes fast communication critical to efficiency, else processor idle time will lead to wasted power. We note that processor power accounts for the majority of our system power and that it is impractical to turn processors on and off for periods on the order of a dozen cycles. Consequently, we see in Figures 3 - 5 that increasing bus width decreases processor idle time, which decreases system power. For FIR, the power begins increasing again at 256 bits because FIR can not take advantage of the increased width.

We further note that Amdahl's law comes into play, and we see the greatest power savings as we initially double bus width, cutting communication latencies in half. As we continue to invest in bus bandwidth, processor idle time becomes a smaller fraction of total run time. With cost as a concern, an area-conscious design philosophy would be to choose a bus width of 64 or 128 bits, where we get the most bang for the buck.

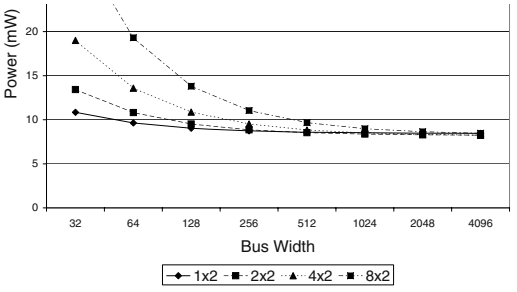


Fig. 5. FFT power as the bus width increases for various tile configurations

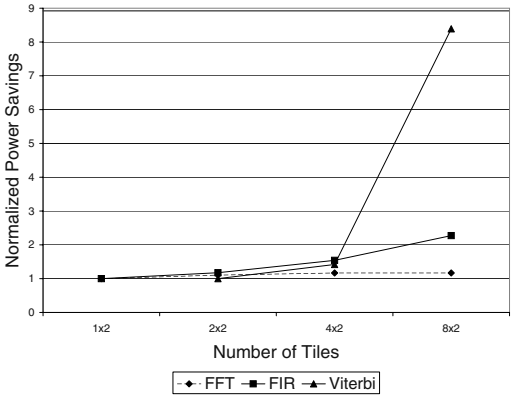


Fig. 6. Power Savings when using Segmented Buses over Unsegmented Buses

5.3 Impact of Segmented Buses

Segmenting the bus allows two simultaneous, short-distance messages to use the same bits in the wire. At the low frequencies of the Synchroscalar system, segmenters are simple transmission gates with little signal restoration or latency involved. Figure 6 shows that as the number of tiles in the column increases, the savings from segmentation also increases, because there are more messages that can traverse the bus at once. Dramatic savings are seen in Viterbi with 8x2 tiles. Even at 4x2, the applications observe 17-54% power savings.

5.4 Discussion

Our simple design-space exploration has revealed several results that challenge our intuitions of microprocessor design. Primarily, substantial global intercon-

nect makes sense in this domain. Low operating frequencies allow signals to traverse global buses in a single cycle. Data dependencies and tile power make the latency of global communication critical. Furthermore, statically-scheduled segmented buses allow the power and utilization of our interconnect to approximate more specialized interconnects as used in ASICs.

6 Related Work

The challenges presented by next generation applications in terms of higher data rates, lower power requirements, shrinking time-to-market requirements, and lower cost has resulted in a tremendous interest in architectures and platforms for embedded communication appliances in the past few years. Researchers have approached the problem from several different angles. The DSP architecture companies have proposed highly parallel VLIW machines coupled with hardware accelerators or co-processors for the computation-intensive functions. The TI OMAP is a good example of this category of solutions. The programmable logic community has been very active in this area, as well, and there are numerous architectural proposals that are derivatives of the standard FPGA. The SCORE project at UC Berkeley [5] and the PipeRench project at CMU [16] are especially noteworthy. They use the dynamic reconfigurability of field-programmable gate arrays to exploit power and performance efficiency. The PLEIADES project at UC Berkeley [21] proposes an interconnection of a low power FPGA, datapath units, memory, and processors, optimized for different application domains. The Pleiades researchers conclude that a hierarchical generalized mesh interconnect structure [22] is most appropriate for their architecture because it balances both the global and the local interconnect. Our results are in agreement with this conclusion in general but given that we are targetting streaming computations such as those encountered in a wirless transceiver, we have greater emphasis on near-neighbor communication, so we have stayed away from a general mesh.

The adaptive SOC project at University of Massachussets [10] advocates an array of processors connected by a statically scheduled communication fabric. They allow different processors to operate at different clock frequencies and demonstrate significant power savings on video processing benchmarks. The key differences between this work and Synchronscalar are in the structure and contents of the tiles and the memory architecture. In aSOC the tiles are hardwired functional blocks such as Viterbi decoder, FFT, DCT etc., while in Synchroscalar we assume programmable DSPs as the building blocks for the tiles. As a result, the memory architecture of the system is radically different, changing the data transfer and communication scheduling problem as well. But, it would be interesting to compare the results between the Synchroscalar and aSOC approaches.

Recently, there has been a revival of interest in locally synchronous and globally asynchronous (GALS) approach to processor implementation [1] including the use of multiple clock domains and multiple voltages [11] [17]. The key difference between GALS approach and the Synchroscalar approach is the restriction of using only rationally related frequencies between different columns. This

avoids the use of asynchronous FIFOs with their synchronization overhead. So, synchroscalar is similar to Numesh [18], rather than the GALS approach.

Synchroscalar's use of spatial rather than temporal flexibility is somewhat inspired by the MIT RAW project [20] [9], but our focus on low power and embedded applications is significantly different. Nevertheless, we expect to be further inspired by the extensive compiler work from the RAW group. Although their compiler algorithms are geared towards dynamic general microprocessor algorithms such as speculation and caching, we expect to leverage their experiences with program analysis and resource allocation.

Another project with a less embedded focus is the Imagine stream processor, a tile architecture at Stanford [13]. Their experience with streaming applications will also be invaluable to the design of our high-level software. Our emphasis on Synchroscalar regions for power reduction and static scheduling of rationally-clocked communication, however, will add significant challenges to our software solutions. Furthermore, both Imagine and RAW are focused on large-system scalability rather than the inexpensive design points of small, embedded systems. We believe that Synchroscalar's differing focus in cost and power will lead to significantly new tradeoffs and design decisions.

7 Conclusion

The goal of this work was to guide the initial design of tile-based embedded architecture. Through simple power models, we found that our original intuitions regarding interconnect did not apply to the low-frequency, data-dependent nature of our application domain. We found that wide, segmented global buses give us some of the low latency and flexibility that conventional DSPs lack. We plan to continue our evaluation of the Synchroscalar architecture through extensive design and simulation of end-to-end applications. We are confident that a novel architecture can meet the challenges of tomorrow's embedded applications.

Acknowledgements

This work is supported by NSF ITR grants 0312837 and 0113418, and NSF CAREER and UC Davis Chancellor's fellowship awards to Fred Chong.

References

1. B. M. Baas. A parallel programmable energy-efficient architecture for computationally intensive DSP systems. In *Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems, and Computers*, Nov 2003.
2. S. Bhattacharya, P. Murthy, and E. Lee. Software synthesis from dataflow graphs, 1996.

3. S. Bhattacharya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing*, (21):151–166, June 1999.
4. J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.
5. E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *Field-Programmable Logic and Applications, FPL-2000*, pages 605–614, 2000.
6. R. Ho, K. Mai, and M. Horowitz. The future of wires. In *Proceedings of the IEEE*, volume 89, pages 490–504, April 2001.
7. R. Kolagotla, J. Fridman, B. Aldrich, M. Hoffman, W. Anderson, M. Allen, D. Witt, R. Dunton, and L. Booth. High Performance Dual-MAC DSP Architecture. *IEEE Signal Processing Magazine*, July 2002.
8. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1), January 1999.
9. W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.
10. J. Liang, S. Swaminathan, and R. Tessier. aSOC: A scalable, single-chip communications architecture. In *IEEE PACT*, pages 37–46, 2000.
11. D. Marculescu and A. Iyer. Power and performance evaluation of globally asynchronous locally synchronous processors. In D. DeGroot, editor, *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-02)*, volume 30, 2 of *Computer Architectuer News*, pages 158–170, New York, May 25–29 2002. ACM Press.
12. T. Mori, B. Amrutur, M. Horowitz, I. Fukushi, T. Izawa, and S. Mitarai. A 1v 0.19mw at 100 mhz 2kx16b sram utilizing a half-swing pulsed-decoder and write-bus architecture in 0.25 μm dual-vt cmos. In *Solid-State Conference, 1998, Digest of Technical Papers, 45th ISSCC 1998 IEEE International*, 1998.
13. S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *International Symposium on Microarchitecture*, pages 3–13, 1998.
14. T. Sakurai and R. Newton. Alpha-Power Law MOSFET Model and Its Application to CMOS Inverter Delay and Other Formulas. *IEEE Journal of Solid State Circuits*, 25:584–594, April 1990.
15. L. Sarmenta, G. A. Pratt, and S. Ward. Rational clocking. In *International Conference on Computer Design*, pages 271–278, 1995.
16. H. Schmit, S. Cadambi, M. Moe, and S. Goldstein. Pipeline reconfigurable FPGA. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 24(2):129–146, March 2000.
17. G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA*, pages 29–42, 2002.
18. D. Shoemaker, F. Honore, C. Metcalf, and S. Ward. Numesh: An architecture optimized for scheduled communication. *Journal of Supercomputing*, 10(3), 1996.
19. M. Y. T. Kumura, M. Ikekawa and I. Kuroda. VLIW DSP for Mobile Applications. *IEEE Signal Processing Magazine*, July 2002.

20. M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, Mar./Apr. 2002.
21. H. Zhang, V. Prabhu, V. George, M. Benes, A. Abnous, and J. Rabaey. A 1-V heterogenous reconfigurable DSP IC for wireless baseband digital signal processing. *IEEE Journal of Solid State Circuits*, 35:1697–1704, November 2000.
22. H. Zhang, M. Wan, V. George, and J. Rabaey. Interconnect Architecture Exploration for Low Energy Reconfigurable Single-Chip DSP. In *Proceedings of the Workshop on VLSI, Orlando, Florida*, April 1999.
23. Y. Zhang, W. Ye, and M. J. Irwin. An alternative for on-chip global interconnect: Segmented bus power modeling. In *Conference Record of the Thirty-Second Asilomar Conrference on Signals, Systems and Computers*, pages 1062–1065, 1998.

Heterogeneous Wireless Network Management

Wajahat Qadeer[1], Tajana Simunic Rosing[1,2], John Ankcorn[2],
Venky Krishnan[2], and Givanni De Micheli[1]

[1] Stanford University, Gates Computer Science Building,
Stanford, CA
{wqadeer, tajana, nanni} @stanford.edu
[2] HP Labs, 1501 Page Mill Road MS 1181
Palo Alto, CA 94304
{tajana, jca, venky}@hpl.hp.com

Abstract. Today's wireless networks are highly heterogeneous, with mobile devices consisting of multiple wireless network interfaces (WNICs). Since battery lifetime is limited, power management of the interfaces has become essential. We develop an integrated approach for the management of power and performance of mobile devices in heterogeneous wireless environments. Our policy decides which WNIC to employ for a given application and optimizes its usage based on the current power and performance needs of the system. The policy dynamically switches between WNICs during program execution if data communication requirements and/or network conditions change. We have experimentally characterized Bluetooth and 802.11b wireless interfaces. Our policy has been implemented on HP's IPAQ portable device communicating with HP's HotSpot server [14]. The applications we tested range from MPEG video to email. The results show that our policy offers a large improvement in power savings as compared to singly using 802.11b or Bluetooth while enhancing performance.

1 Introduction

Mobile communications today has heterogeneous wireless networks providing varying coverage and QoS. Portable devices typically have more than one type of wireless interface built-in. For example, most recent IPAQs have 802.11b, Bluetooth and ability to add a GPRS PCMCIA card. To satisfy the bandwidth and QoS constraints of the applications, the mobile devices need to be able to seamlessly switch among their wireless network interfaces.. Additionally, the high communication and computation cost of applications is a burden on the battery life of portable devices. Capacity of a battery has not increased tremendously. Improvements of only a factor of 2-4 have been observed during the past 30 years. The ever-increasing need for battery lifetime in mobile devices demands a tighter control over its energy consumption.

Although low-power circuit design forms the basis of power management in a mobile device, higher-level management of power dissipation offers many more advantages. These techniques allow seamless integration between user applications and power management policy design thus allowing energy consumption to be reduced

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 86–100, 2004.

while maintaining a desired QoS. During program execution communication interfaces are placed in low-power states depending upon their acces patterns and application performance needs.

The techniques developed to date for the enhancement of heterogeneous networks concentrate on improving their accessibility and QoS. These methods enable mobile devices to communicate with each other by introducing changes in the network protocol stack. They also allow establishment and maintenance of connections between mobile hosts using available links to improve robustness and performance. However, none of the techniques adequately addresses power management. Power reduction methodologies presented in the past largely focus on improving energy consumption of one single device.

This work presents a new methodology for managing power and performance of mobile devices consisting of heterogeneous WNICs. The policy formulated decides what network interface to employ on a portable device for a given pattern of usage. The decision is governed by the current power dissipation and QoS requirements of the system. The maximum likelihood estimator is employed for tracking system changes. It detects variations in the average throughput of available wireless interfaces and the data usage patterns. The policy for power and performance management (PPM) decides what wireless network interface card to use, what low-power state to employ , the transition times between active and low-power states and the buffer size to use for good application QoS. We implemented the policy on HP's IPAQ portable device that is communicating with HP's HotSpot server [14] via Bluetooth and 802.11b. The applications we tested range from MPEG video to email. Our results show both large savings in power when using a single WNIC, as well as seamless switching with concurrent power savings among WNICs.

2 Related Work

Mobile devices require wireless communication interfaces to facilitate connectivity with Internet and with the other devices. A mechanism is required for forwarding packets between different wireless networks due to increasing device mobility. Mobile IP [1] provides one example of such mechanism. Changes are introduced in the network and link layers of the network protocol stack that assist the host's home network in forwarding packets to its network of residence. However, with mobile IP even if communicating devices are in the same wireless network, data needs to traverse a multi-hop path. In order to perform localized communication between devices, which are one hop distance away, *Contact Networking* [2] has been proposed. By allowing seamless switching between multiple diverse interfaces, this technique enhances robustness and QoS of the network.

Mobile hosts experience varying data rates during communication in part due to lossy nature of the wireless link. In order to avoid disruptions, a distributed file system has been developed [3], [4]. It allows application aware and application independent adaptation to a temporary loss or degradation of the wireless link thus enhancing robustness. A method for improving hand-offs proposes buffering data

on multiple base stations in close proximity to the mobile host [5] thus achieving seamless switching between base stations. Telephony and data services spanning diverse access networks have been integrated in [6]. However, the focus of these techniques has been on the enhancement of performance and QoS of heterogeneous networks. Power management of communicating hosts has been mainly overlooked.

Several techniques have been proposed to efficiently manage power dissipation in portable devices. These methods employ diverse mechanisms to predict periods of inactivity during communication. Based upon these predictions the mobile device is put into a low-power state. The most basic power management policy is a time-out. If the device remains idle for a certain period, it is put into a low power state. Similarly, a device can enter low-power mode when idleness is being anticipated in a connection [7]. However, incorrect estimates cause performance and power penalties. In contrast, stochastic models derive provably optimal power management policies. *Pure Markov decision processes* [8], [9] employ either discrete or continuous time memory-less distributions. However, discrepancies have been observed in predicted and actual power savings owing to history dependent nature of real world processes. *Time-indexed semi Markov decision processes* [10] are based upon history based distributions. This technique has demonstrated energy savings in real-world applications. The power management techniques presented to date mostly focus on the reduction of power dissipation in one WNIC. This leads to inefficient power management for portables with multiple diverse communication interfaces.

Methods being employed for the performance enhancement of homogeneous networks put a lot of emphasis on power management. IEEE 802.11 [11] standard implements power management by sending a traffic indication map (TIM) with the beacon to the client. It enables the client to enter doze mode if no more data is available. Since the device still has to wake up after every beacon interval for TIM, a new technique proposes decoupling of control and data channels [12]. The control channel uses low-power radio and wakes up the device whenever data is present. A survey of energy efficient network protocols for wireless is presented in [19]. Application level information is used for power management in [13]. In our work we developed an integrated policy for power and performance management. Our power and performance management (PPM) algorithm dynamically selects the appropriate wireless network interface with the goal of minimizing the overall energy consumption while meeting application's QoS requirements. We present measurement results that show large energy saving with good QoS while using Bluetooth and 802.11b on HP's IPAQ for a typical set of applications.

The rest of this work is organized as follows. Section 3 discusses the characterization of Bluetooth and 802.11b interfaces. Details of the heuristic policy for choosing among network interfaces are presented in Section 4 whereas the results and conclusion are discussed in Sections 5 and 6 respectively.

3 Characterization of Devices

3.1 Bluetooth

Bluetooth has been developed as a radio link with a short range to provide wireless connectivity to portable and fixed devices. It operates in 2.4GHz ISM band. Bluetooth supports point-to-point and point-to-multi-point connections called *piconets*. A piconet can consist of two to eight active Bluetooth devices. One device is the master and the rest are its slaves. In addition, a master can support several other inactive slaves, which have been *parked*. These slaves remain synchronized to the master but do not become a part of the piconet. A s*catternet* is composed of multiple piconets with an overlapping coverage area. Bluetooth provides both synchronous and asynchronous connections. Data rate and the average power consumption are a function of the packet type selected. Bluetooth state space is shown in Figure 1.

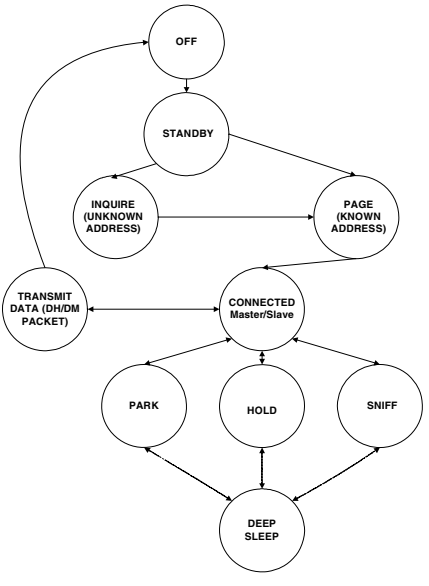


Fig. 1. Bluetooth state space

Three low-power states are supported in Bluetooth standard: park, hold and sniff. They can be activated once a connection exists between Bluetooth devices. Transition times and average power dissipation for switching between the modes are shown in Table 1. The CSR Bluetooth chips also supports *deep sleep* state with only 270uW power consumption [15]. Deep sleep state can be entered only if there is no activity on UART for at least 250ms. The main characteristics of the three low-power states are outlined below.

- Hold mode is employed to stop data transfer by the requested device for a negotiated interval. It is especially useful if the requesting device wants to perform inquiry, page and scan or burst mode transfer operation in scatternets.
- Sniff mode is useful on low data rate links such as email where a quick response is required whenever data is present. During an attempt window the device looks for any incoming data. If no data is present, it goes into low-power mode; however, if data is present, the device listens to the master for the specified time-out period. Sniff mode can also be useful in scatternets for devices that are a part of two piconets.
- Park mode is used to enhance the number of simultaneous connected slaves. As link set up takes about 10s in Bluetooth, it is best to retain an established connection. In this mode no data transfer takes place as the parked slave gives up its connection ID but it remains synchronized to the master.

Table 1. Bluetooth low-power mode measurements

	Transition time (ms)	Avg. power (W)
Active Mode		0.09 – 0.24
Hold Mode		0.061
Hold mode entry	1.68	0.068
Hold mode exit	11.62	0.216
Park Mode		0.061
Park mode entry	2.16	0.077
Park mode exit	4.12	0.126
Sniff Mode		0.061
Sniff mode entry	0.94	0.078
Sniff mode exit	7.36	0.194

Bluetooth supports multiple packet types for both asynchronous and synchronous connections. These packet types differ in data payload size and error correction algorithms. Maximum achievable throughput for various packet types is tabulated in Table 2. Our measurement results come close to throughput values reported in Table 2.

Table 2. Maximum throughput for Bluetooth ACL connection

Packet name	Symmetric max rate (kb/s)	Asymmetric max rate (kb/s)	
		Forward	Reverse
DM1	108.8	108.8	108.8
DM3	258.1	387.2	54.4
DM5	477.8	477.8	36.3
DH1	172.8	172.8	172.8
DH3	390.4	585.6	86.4
DH5	433.9	723.2	57.6

For instance, the maximum and the average throughput numbers were measured on CSR Bluetooth for DH5 packets at 87kB/s and 79kB/s respectively. The throughput increases with an increase in the payload capacity of the base-band packet. However, throughput can significantly decreased in the presence of noisy channels if less error-correction bits are present. The range of Bluetooth devices is enhanced by an increase in transmission power; but that causes a further energy drain from the battery.

3.2 Wireless LAN – 802.11b

802.11b has been developed to provide fast wireless connectivity to mobile devices. Theoretically, 802.11b can support a maximum data rate of 11Mbps in 2.4GHz band. It is designed to work in adhoc as well as infrastructure network topologies. Today a large majority of all WLAN communication happens in infrastructure mode, thus this is the mode we will be focusing on in this work. An access point acts as a bridge between wired and wireless networks. An association is developed between the access point and the 802.11b card before commencing data communication. In order to facilitate mobility, access-points also support roaming.

WLAN has two active states, transmit and receive, in addition to two low-power modes, doze and off. Table 3 shows average power dissipation measurements for the above mentioned power states. According to the 802.11b standard, a synchronization beacon is transmitted to the awake card by a central access point (AP) every 100ms. The beacon is followed by a traffic indication map (TIM) indicating any required data transfers. Doze mode is activated until the next beacon if no data transfer is required. This power management (PM) policy does not always give optimal power savings due to three main factors. First, an increasing number of clients causes radios to stay on longer since there is more contention due to multiple simultaneous synchronization attempts by the mobiles. Second, 802.11b's response time to the AP suffers due to the delays imposed by the doze mode. Finally, even without any running applications, 802.11b spends a considerable amount of time listening with an increase in broadcast traffic and is thus unable to enter doze mode. Doze mode can only be activated by the hardware. Transitions to the off state from either the active or the doze state can be controlled at the OS level. Transition times and average power dissipation for switching between active and low-power states have been tabulated in Table 3 for Cisco 350 WNIC.

Table 3. 802.11b low-power mode measurements

	Transition time (ms)	Avg. power (W)
Transmit state	-	2.25
Receive state	-	1.4
Doze state	-	0.75 – 1.4
Doze state entry	0.1	1.4
Doze state exit	1	1.6
Off state	-	
Off state entry	1	1.7
Off state exit	300	2.3

4 Power and Performance Management

The goal of PPM is to enhance QoS while minimizing power dissipation in a portable device. PPM's primary task is to determine what network interface is most suitable for the application needs and how to manage its power and performance states. When an application starts on a portable device, PPM pre-selects those WNICs for data communication whose average throughput is greater than the data consumption rate of the application. This ensures that the QoS requirements of the application are satisfied. In streaming applications a special emphasis is placed upon the data buffer size. It not only determines the average sleep time of the communication device but also the energy dissipated in the RAM. Since the size of the buffer is determined by the difference between the throughput and the data consumption rate, all the pre-selected WNIC are further examined to determine not only their communication power dissipation but also the resulting RAM power consumption. The one that offers maximum power savings is selected. Additionally, during the examination of the communication energy only those low-power states, which are most suitable for the current scenario, are considered. PPM also defines the switching time between active and sleep states for the selected WNIC.

$$\ln(P_{\max}) = (w-c+1)\ln\frac{\lambda_{new}}{\lambda_{old}} - (\lambda_{new}-\lambda_{old})\sum_{j=k}^{m}\Delta t_j \tag{1}$$

PPM dynamically keeps track of the variations in the application data consumption rate and the throughput of wireless interfaces using the log of the maximum likelihood estimator as shown in equation (1). A change in rate is defined to occur at point c when computed likelihood over the last w data points is greater than a preset threshold. In our work we use 99.5% as a threshold. The change is observed between the old, λ_{old}, and the new rate, λ_{new}. Details of this algorithm are further discussed in [17]. Whenever a change occurs, the PPM evaluates which subset of WNICs could handle the applications currently running by insuring that the available WNIC's throughput rate, λ_t, is greater than the application's data consumption rate, λ_u.

$$\lambda_t \geq \lambda_u \tag{2}$$

Network interfaces satisfying equation (2) are further analyzed to identify the interface that offers maximum power savings for the given application while keeping the required quality of service. The total energy, E_{total}, consumed during a given session along with the average power dissipation, P_{avg}, is given by the following equations:

$$E_{total} = E_{comm} + E_{RAM} + P_{switch}T_{switch} \tag{3}$$

$$P_{avg} = E_{total} / \left(\left(B_{active} / \lambda_u \right) + T_{switch} \right) \tag{4}$$

Where E_{comm} and E_{DRAM} denote the energy consumed by the WNIC and the DRAM respectively during the communication period, B_{active} specifies the size of the buffer

for streaming applications that is actively read and written to in steady state. P_{switch} and T_{switch} indicate the average power dissipated and the time taken when switching from one WNIC to another. Only WNICs with throughput high enough to meet applications demands are considered. Figure 2 depicts the process of switching interfaces. The details of this procedure are further elaborated in [2] and [18].

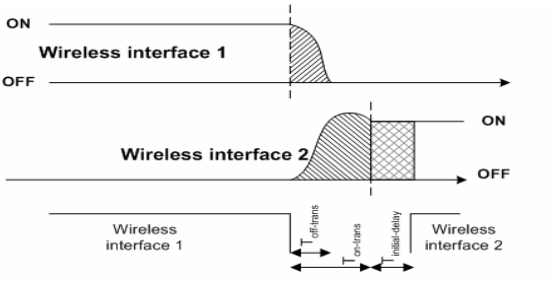


Fig. 2. Procedure for switching wireless network interfaces

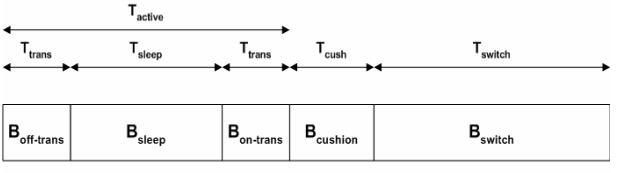


Fig. 3. Buffer layout and the associated time intervals

In streaming applications the size of the buffer directly affects E_{comm} and E_{DRAM}. If the size of the buffer undergoes an increase, the average power dissipation of the communication device diminishes due to longer sleep periods and thus less overhead in transition between power states. On the other hand, the RAM energy increases with increasing buffer sizes as the number of active banks increases. Thus the determination of the buffer size is of principal importance for enhancing power savings. The size of the buffer is chosen in a way such that the transmissions could be scheduled in bursts. In between bursts the WNIC can transition into low power mode, thus saving energy and freeing bandwidth from contention. The size of the buffer is determined according to the equations given below.

$$B = B_{active} + B_{switch} + B_{cush} \tag{5}$$

$$B_{active} = B_{sleep} + B_{on-trans} + B_{off-trans} \tag{6}$$

$$B_{sleep} = \left(T_{sleep}\lambda_t\lambda_u\right)/\left(\lambda_t - \lambda_u\right) \tag{7}$$

$$B_{off-trans} = \left(T_{off-trans}\lambda_t\lambda_u\right)/\left(\lambda_t - \lambda_u\right) \tag{8}$$

$$B_{on-trans} = \left(T_{on-trans}\lambda_t\lambda_u\right)/\left(\lambda_t - \lambda_u\right) \tag{9}$$

$$B_{switch} = T_{switch}\lambda_u \tag{10}$$

$$B_{cush} = T_{cush}\left(\lambda_t\left(1-\chi\right) - \lambda_u\left(1+\delta\right)\right) \tag{11}$$

$$\left(T_{be}\lambda_t\lambda_u\right)/\left(\lambda_t - \lambda_u\right) \le B \le B_{\max} \tag{12}$$

Where:

- T_{sleep} is the average sleep interval of the communication device.
- T_{switch} accounts for the worst case delay encountered in dynamically switching between two network interfaces.
- T_{cush} provides a cushion for any small variations present in the system.
- χ and δ denote small variations in throughput and data consumption rate respectively.
- $T_{off\text{-}trans}$ and $T_{on\text{-}trans}$ are the transition times between active and low-power and low-power and active states respectively.
- T_{be} is the break-even time and is defined in terms of power consumed during the transition, $P_{trans} = P_{on\text{-}trans} + P_{off\text{-}trans}$, the power consumed in the active and sleep states, P_{active} and P_{sleep}.

$$T_{be} = T_{trans} + T_{trans}\frac{P_{trans} - P_{active}}{P_{active} - P_{sleep}} \tag{12}$$

- B_{max} is the maximum amount of memory available.

When an application starts, it waits for $T_{initial\text{-}delay}$ before beginning to read from the buffer. This time interval is influenced by the maximum delay a user can tolerate at start-up. Thus the time to enter steady state, $T_{steadystate}$, is given by equation 14. During this interval the communication device stays in the active mode.

$$T_{steadystate} = T_{initial-delay} + \left(B - T_{initial-delay}\lambda_t\right)/\left(\lambda_t - \lambda_u\right) \tag{13}$$

The total energy consumed by the communication device, E_{comm}, during the buffer refill period is given by equation (15). Note that for simplicity the transition power and time have been combined into one variable, P_{trans} and T_{trans}. The communication energy needs to be balanced by the energy consumed by memory, as larger buffer sizes cause higher energy consumption.

$$E_{comm} = P_{active}\left(T_{active} + T_{cushion}\right) + P_{trans}T_{trans} + P_{sleep}T_{sleep} \tag{14}$$

The amount of energy consumed by memory, E_{RAM}, is determined from the energy consumed by the banks that are actively participating in reading and writing of data, E_{active}, and the energy of non-active banks, $E_{non\text{-}active}$:

$$E_{RAM} = E_{active} + E_{non-active} \tag{15}$$

$$E_{active} = P_{write}T_{active} + \left(P_{read} + N_{abanks}P_{refresh}\right)\left(B_{active}/\lambda_u\right) \tag{16}$$

$$E_{non-active} = \left(N_{banks} - N_{abanks}\right)P_{non-active} \tag{17}$$

$$N_{abanks} = \lceil B/Size_{bank} \rceil \tag{18}$$

Where:

- P_{write} and P_{read} specify the average power dissipated when the RAM is written and read respectively.
- N_{banks} specifies the total number of available memory banks.
- $P_{non\text{-}active}$ is the average power consumed by the memory banks that are non-active.
- $P_{refresh}$ is the average power spent in refreshing the active banks not participating in read and write operations.
- N_{abanks} is the number of memory banks where each has size $Size_{bank}$

Figure 4 shows the decrease in communication energy as the buffer size increases whereas the increase in RAM energy with an increase in the buffer size is shown in Figure 5.

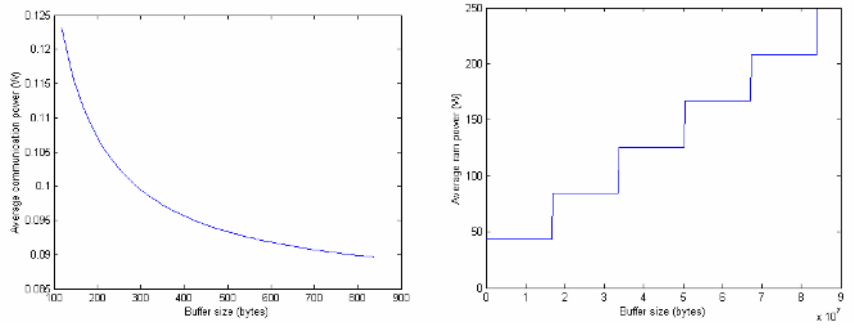


Fig. 4 and 5. Communication (left – Fig. 4) and DRAM power consumption (right – Fig 5.) in terms of buffer size

PPM pre-selects WNICs for a particular application based upon their average throughputs and the data consumption rate of the application. The WNIC that offers

minimum power dissipation with regards to communication and RAM is selected. The appropriate low-power state of the WNIC along with the switching points is also defined by PPM. Additionally, it can dynamically switch the selected WNIC if a change in its throughput and/or the average data consumption rate of the application is detected. In the next section we present the results obtained by using our PPM with a typical set of applications having diverse data usage patterns.

5 Results

Our measurements were performed with a research prototype of HP's HotSpot server [14] and IPAQ 3970 client that supports both 802.11b (CISCO Aironet 350 PCMCIA WLAN) and Bluetooth (CSR) interfaces. The operating system running on the IPAQ is Linux. The power measurements have been performed with a DAQ card at 10ksamples/sec. In our experiments, we have used transmission control protocol (TCP) for all data communication. For Bluetooth this has been done using *bnep.* At first we consider individual applications and employ our PPM to determine the appropriate WNIC for each one based upon its data consumption rate and the average throughput supported by the WNIC. For this experiment we have assumed that the throughput and the data usage pattern do not change significantly during program execution.

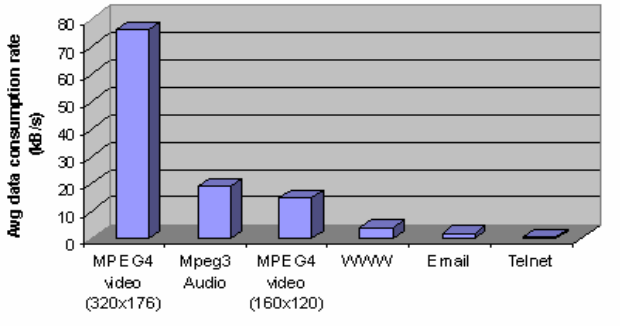


Fig. 6. Application data consumption rate (kB/s)

For MP3 audio streaming Bluetooth and 802.11b offer similar performance, but 802.11b gives more power savings as shown in Figure 7. Note that Bluetooth cannot be turned off as the reestablishment of the connection requires 1-10 sec. In contrast, as shown in Figure 8, Bluetooth is more suitable for email traffic as it offers larger power savings in park mode with deep sleep enabled. 802.11b incurs a significant power dissipation penalty when frequently switching from off to active state.

Bluetooth seems to be the connection of choice for telnet and WWW based applications owing to its faster response time and low-power dissipation as compared to 802.11b. Again the suitable low-power state for Bluetooth is Park with deep sleep enabled. However, due to a significant decrease in the response time for 802.11b with

off mode, the low-power state of choice for 802.11b is PM. The switching overhead associated with the results is shown in Figures 9 and 10.

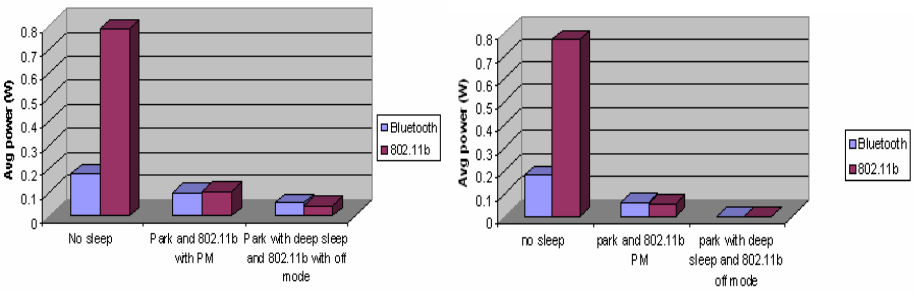


Fig. 7 and 8. WNIC power consumption for MP3 audio (left – Fig.7.) and email (right – Fig. 8.)

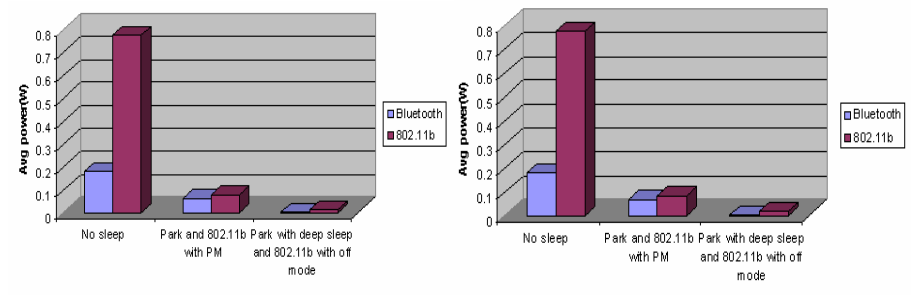


Fig. 9 and 10. WNIC power consumption for Telnet (left – Fig. 9.) and WWW (right – Fig. 10.)

For MPEG2 video streaming, Bluetooth can only be employed for small images due to its lower average throughput. The low-power mode for Bluetooth is again park with deep sleep enabled. However, 802.11b with off mode offers more power savings due to higher throughput and non-existent power dissipation in the off state. The results are shown in Figure 11. Power dissipated by the communication device is an increasing function of the buffer size in streaming applications. However, by using PPM in our particular setup, we found that the buffer size is limited by the power dissipated in the DRAM.

Table 4 shows the percentage of time spent in low-power state for the most optimal power and performance management policy per application. Note that the energy savings for these policies have been persented in Fig. 7-11 above. Clearly, for each application 802.11b spends more time in low-power mode since its bandwidth is significantly higher than Bluetooth's. In our particular case MP3 files we tested with were the highest quality audio, and thus were larger relative to MPEG2 video file. In fact, video had to be of small size (160x120) due to low bandwidth of Bluetooth. As a result, both interfaces spend more time in low-power state with video than audio.

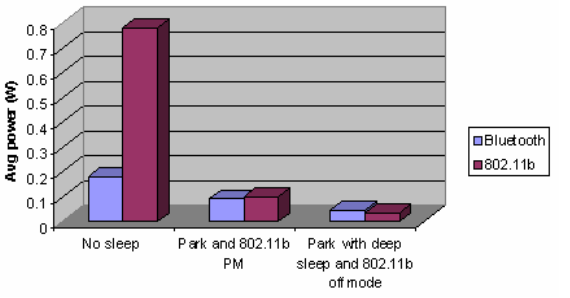


Fig. 11. WNIC power consumption for MPEG video (160x120)

Table 4. Time spent in low-power mode per application type and WNIC

Applications	Bluetooth (%) sleep time	802.11b (%) sleep time
MP3 audio	69.4	96.9
MPEG2 video	76.2	97.6
WWW	92.9	99.3
Telnet	99.3	99.9

Next we present results of the experiment that includes a dynamic switch between wireless interfaces during program execution. In this experiment the data consumption rate of the application stays constant, but the throughput of the selected WNIC undergoes a change. Let's suppose that a person with a portable device is streaming MP3 audio using Bluetooth. After a certain period of time he moves away from the server and the throughput of Bluetooth experiences a sharp decrease. The estimator detects the change in throughput and forces PPM to reevaluate the suitability of the selected network interface. The reevaluation suggests a change in the network interface and 802.11b is chosen over Bluetooth. The comparison of power dissipation is shown in Figure 12. In the comparisan we show that our policy would show significant power savings over just using Bluetooth.

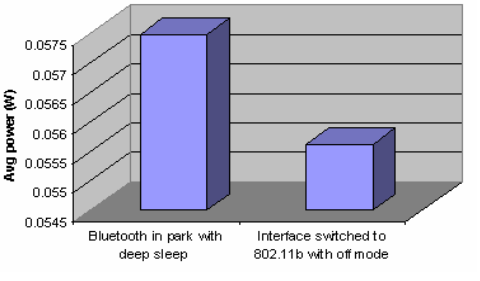


Fig. 12. Throughput change forces a WNIC switch with PPM

Lastly, we analyze the performance of PPM when application data consumption rate changes. We created an application trace consisting of MP3 audio, email, telnet, WWW and MPEG2 video. The trace is executed first by using 802.11b only with PM enabled, secondly by employing Bluetooth only with park as the low-power mode and in the end by PPM with Bluetooth and 802.11b as the available wireless interfaces. When the application trace is executed using 802.11b only, the wireless interface is placed in the doze mode whenever TIM indicates periods of inactivity. Similarly for Bluetooth, the interface is placed in the park mode whenever no data communication is needed. The results are shown in Figure 13. We found that PPM offers a factor of 2.9 times improvement in power savings over just employing Bluetooth with park mode, and a factor of 3.2 times higher than 802.11b with PM the power savings enabled. Moreover, PPM enhances the QoS since wireless interfaces are switched to match the data usage pattern of the application.

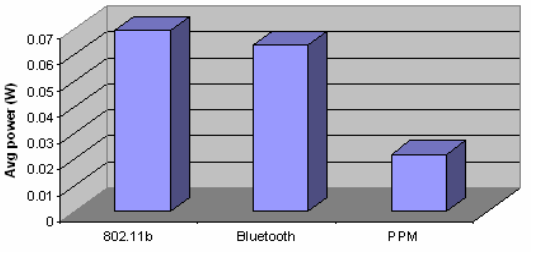


Fig. 13. Change in data consumption rate forces a WNIC switch when using PPM

6 Conclusion

This work presents a new methodology for enhancing QoS while maximizing power savings in heterogeneous wireless systems. A policy for selecting the most appropriate network interface for a particular application has been developed. We have tested our policy on IPAQ 3970 supporting 802.11b and Bluetooth wireless interfaces using various typical applications. We have shown that our PPM offers 2.9 and 3.2 power savings over solely using Bluetooth and 802.11b respectively when running a string of applications including MPEG video and MP3 audio.

References

[1] D. B. Johnson, and D. A. Maltz, "Protocols for adaptive wireless and mobile networking," IEEE Personal Communications, 1996.

[2] C. Carter, R. Kravets, and J. Tourrilhes, "Contact Networking: A Localized Mobility System," in Proc. of the First International Conference on Mobile Systems, Applications, and Services, 2003.

[3] J. Flinn, M. Satyanarayanan, "Energy-aware adaptation for mobile applications," Proceedings of the 17th ACM Symposium on Operating Systems Principles, 1999.

[4] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, K. Walker, "Agile Application-Aware Adaptation for Mobility," in Proc. of 16th ACM Symposium on Operating System Principles, 1997.

[5] Eric A. Brewer, Randy H. Katz, Elan Amir, Hari Balakrishnan, Yatin Chawathe, Armando Fox, Steven D. Gribble, Todd Hodes, Giao Nguyen, Venkata N. Padmanabhan, Mark Stemm, Srinivasan Seshan, and Tom Henderson, "A network architecture for heterogeneous mobile computing," IEEE Personal Communications Magazine, 5(5):8 -- 24, 1998..

[6] Helen J. Wang, Bhaskaran Raman, Chen-nee Chuah, Rahul Biswas, Ramakrishna Gummadi, Barbara Hohlt, Xia Hong, Emre Kiciman, Zhuoqing Mao, Jimmy S. Shih, Lakshminarayanan Subramanian, Ben Y. Zhao, Anthony D. Joseph, and Randy H. Katz, " ICEBERG: An Internet-core Network Architecture for Integrated Communications," IEEE Personal Communications: Special Issue on IP-based Mobile Telecommunication Networks, 2000.

[7] C. H. Hwang and A. Wu, "A Predictive System Shutdown Method for Energy Saving of Event-Driven Computation", in International Conference on Computer Aided Design, pp. 28–32, 1997.

[8] E. Chung, L. Benini and G. De Micheli, "Dynamic Power Management for non-stationary service requests", Design, Automation and Test in Europe, pp. 77–81, 1999.

[9] Q. Qiu and M. Pedram, "Dynamic power management of Complex Systems Using Generalized Stochastic Petri Nets", Design Automation Conference, pp. 352–356, 2000.

[10] T. Simunic, L. Benini, P. Glynn, G. De Micheli, "Event-Driven Power Management," IEEE Transactions on Computer-Aided Design, July 2001.

[11] The Editors of IEEE 802.11, IEEE P802.11D5.0 Draft Standard for Wireless LAN, July, 1996.

[12] E. Shih, P. Bahl, M. J. Sinclair, "Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices," in Proc. of ACM MobiCom, 2002.

[13] A. Acquaviva1, T. Simunic, V. Deolalikar, and S. Roy, "Remote power control of wireless network interfaces," in Proc PATMOS, 2003.

[14] D. Das, G. Manjunath, V. Krishnan, P. Reddy, "HotSpot! – a service delivery environment for Nomadic Users System Architecture," Mobile Systems and Storage Lab., HP Laboratories Palo Alto, CA Rep. HPL-2002-134, 2002.

[15] CSR, "BlueCore Power Saving Modes," Jan, 2003.

[16] Bluetooth Special Interest Group, "Specification of the Bluetooth System 1.1, Volume 1: Core," http://www.bluetooth.com, Feb, 2001.

[17] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli, "Dynamic voltage scaling for portable systems," in Proceedings of the 38th Design Automation Conference, June 2001.

[18] J. Tourrilhes and C. Carter, "P-Handoff: A protocol for fine-grained peer-to-peer vertical handoff," Proc. of PIMRC, 2002.

[19] C. E. Jones, K. M. Sivalingam, P. Agrawal, and J. C. Chen. "A survey of energy efficient network protocols for wireless networks", *Wireless Networks*, 7(4):343–358, July 2001.

"Look It Up" or "Do the Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization

Daniel Citron[1] and Dror G. Feitelson[2]

[1] IBM Haifa Labs,
Haifa University Campus,
Haifa 31905, Israel
citron@il.ibm.com
[2] School of Computer Science and Engineering,
The Hebrew University of Jerusalem,
91904 Jerusalem, Israel
feit@cs.huji.ac.il

Abstract. Instruction reuse and memoization exploit the fact that during a program run there are operations that execute more than once with the same operand values. By saving previous occurrences of instructions (operands and result) in dedicated, on-chip lookup tables, it is possible to avoid re-execution of these instructions. This has been shown to be efficient in a naive model that assumes single-cycle table lookup. We now extend the analysis to consider the energy, area, and timing overheads of maintaining such tables.

We show that reuse opportunities abound in the SPEC CPU2000 benchmark suite, and that by judiciously selecting table configurations it is possible to exploit these opportunities with a minimal penalty. Energy consumption can be further reduced by employing confidence counters, which enable instructions that have a history of failed memoizations to be filtered out. We conclude by identifying those instructions that profit most from memoization, and the conditions under which it is beneficial.

Keywords: Memoization, Reuse, Energy, Area, Lookup.

1 Introduction

During program execution there are operations that execute, more than once, with the same operand values. Several papers published in the late 90s proposed exploiting this fact by saving previous occurrences of instruction level operations (operands and result) in dedicated, on-chip, lookup tables. It is then possible to avoid execution of these instructions by matching the current executing instruction's operands with an entry in the table.

The approach of Sohi and Sodani [7] is to reuse instructions (identifiable by the Program Counter) early in the pipeline by matching their operands or by establishing that their source registers haven't been overwritten since the instruction's last invocation. Their technique is called *Instruction Reuse*. Citron, Feitelson, and Rudolph [8] extend the idea of Richardson [9] and perform the reuse test on the operand values and operation,

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 101–116, 2004.

Table 1. Latencies and throughputs of instructions on current and previous generations of microprocessors

Processor	*Clock Rate*	*IALU*		*IMUL*		*IDIV*		*FADD*		*FMUL*		*FDIV*	
		lt	tp	lt	tp	lt	tp	lt	tp	lt	tp	lt	tp
POWER3-II [1]	450MHz	1	1	5	1	37	37	3	1	3	1	18	18
POWER4 [2]	1.5GHz	1	1	7	6	68	67	6	1	6	1	30	30
Pentium III [3]	1.4GHz	1	1	4	1	56	56	3	1	5	2	38	38
Pentium 4 [4]	3.2GHz	.5	.5	15	5	56	23	5	1	7	2	38	38
UltraSPARC II [5]	480MHz	1	1	5	5	36	36	3	1	3	1	22	22
UltraSPARC III [6]	1.2GHz	1	1	6	5	39	38	4	1	4	1	20	17

in parallel to instruction execution. This enables different static instruction instances to reuse each other's results. This was coined *Instruction Memoization*. Molina, González, and Tubella [10] combine both approaches: a match is first attempted when indexed by the PC and if that fails the operand values are used as an index. A limited study (multiplication in four applications) by Azam, Franzon, and Liu [11] suggests memoization as a power saving method.

However, these models are naive and outdated. They assume that the latency of the table lookup time is a single cycle, that it can be performed in parallel or ahead of computation without any timing or power penalty, and that a successful lookup will enhance performance. These and other shortcomings have been reviewed by Citron and Feitelson [12]. Table 1 shows that the latencies of most instructions on the current generation of microprocessors are growing in comparison to their predecessors. Instruction Memoization, or IM (this is the term we will use throughout this study), has the potential to reduce these latencies and enhance performance given a model that is adapted to the deep pipelines, short cycles, and tight energy budgets of present and future microprocessors. The contributions of this study are fourfold:

1. Ratify that reuse opportunities still exist in CPU intensive applications. Sect. 2 presents the reuse rates for the SPEC CPU2000 suite compiled for IBM's POWER4 [13] 64-bit architecture.
2. Explore the organization of the lookup tables in terms of reuse rate, access time, energy consumption, and area. Sect. 3 performs this analysis with 2^k and full factorial designs.
3. Enhance the reuse process. Sect. 4 will show how using multiple lookup tables (per instruction class), trivial computation detection, and confidence estimators can raise the reuse rate and reduce the miss penalty.
4. Determine which instructions benefit most from IM. Sect. 5 compares the physical features of various functional units to lookup tables that store their results.

This study is just a first step in exploring the power/performance potential of using IM in a modern processor. The final, cycle accurate, performance evaluation has been reserved for future work. We believe that including it in this work would dominate the analyses presented and detract from their impact.

Table 2. Benchmarks (CINT2000 top half, CFP2000 bottom half), input sets, and instructions executed. The default inputs are the *lgred* sets of MinneSPEC. Benchmarks were terminated after 2 billion instructions

Benchmark	*Input*	*# inst.*	*Benchmark*	*Input*	*# inst.*
164.gzip	lgred - log	434M	168.wupwise	lgred	2000M
175.vpr	lgred - place	2000M	171.swim	lgred	304M
176.gcc	lgred	2000M	172.mgrid	lgred	94M
181.mcf	lgred	836M	173.applu	lgred	66M
186.crafty	lgred	838M	177.mesa	lgred	850M
197.parser	lgred	2000M	178.galgel	lgred	210M
252.eon	lgred - cook	761M	179.art	lgred	2000M
253.perlbmk	lgred	1921M	183.equake	lgred	871M
254.gap	lgred	772M	187.facerec	lgred	356M
255.vortex	lgred	1278M	188.ammp	lgred	1207M
256.bzip2	lgred - source	1759M	189.lucas	lgred	212M
300.twolf	lgred	925M	191.fma3d	lgred	540M
			200.sixtrack	lgred	1329M
			301.apsi	lgred	257M

2 Instruction Memoization and Reuse Potential

This section will reconfirm the potential of IM by measuring the reuse rate of an infinitely large lookup table. A lookup table, which we will coin a Memo-Table, is a cache like structure that is composed of a relatively large tag (opcode + operands) portion and a relatively small data portion (result). Fig. 1 shows a Memo-Table designed to contain the opcodes, operands, and results of IBM's POWER4 [13] 64-bit instructions. The extensive use of FMADD (Floating point Multiply ADD) instructions necessitates the storage of three 64-bit operands in the table. The opcode field is composed of 6 bits of the basic opcode (OPC) and 10 bits of the extended opcode (XO). All bits that aren't used are zeroed when an instruction is placed in the table. There is no need for a valid bit, an illegal opcode loaded at boot-time will prevent matching and reading invalid data. The daunting problem of matching 207 bits is one aspect that has been neglected by previous studies that focused on two 32-bit operands. Widespread 64-bit computing and

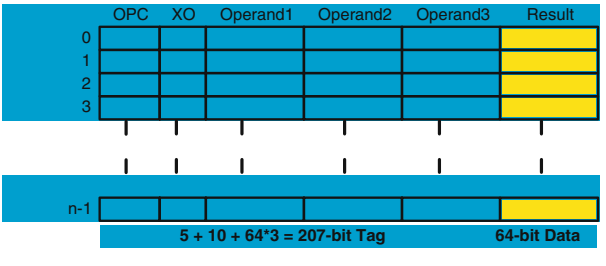


Fig. 1. A generic Memo-Table capable of memoizing all POWER4 instructions

enhanced instruction sets forces us to deal with this problem head-on. Sect. 3 examines the impact this has on performance, power, area and access time.

2.1 Simulation Methodology

The infrastructure for all our simulations is Aria [14], an environment for PowerPC microarchitecture exploration. The environment dynamically traces all user and library code (system calls are executed but not traced). Drivers can be written that collect and analyze any subset of instruction types, data values, memory references etc. Specifically, we built drivers to collect memoization statistics for various instruction types. The data was collected from the SPEC CPU2000 suite using the MinneSPEC [15] input sets. Table 2 shows the exact inputs used and the number of instructions simulated. The C/C++ benchmarks were compiled on a POWER4 running AIX version 5.1 using the IBM compiler `xlc` v6.0 with the flags: `-q64 -DSPEC_CPU2000_LP64 -O5`. The Fortran benchmarks were compiled using the `xlf` v8.1 compiler with the flags: `-q64 -O5`.

2.2 Instruction Memoization Potential

In order to gauge the potential of instruction memoization we performed an experiment that measures the reuse rate of most instructions using an "infinite" MEMO-TABLE (1 million entries with 64-way associativity, LRU replacement, and indexed using the XOR of the lower bits of the operands and opcodes). However two classes of instructions are omitted:

- **Branches:** Conditional branches in the POWER architecture determine their outcome on precomputed bits in condition registers. Memoizing the instruction to obtain the next PC based on the current PC and condition bits is exactly what the Branch Prediction Unit does, there is no need to duplicate this functionality.
- **Loads/Stores:** Memoization of memory references based on the base address and offset requires storing the effective address and implementing an invalidation mechanism every time data is stored to memory. This reduces the technique to just another level in the memory hierarchy. Nevertheless, the effective address calculation is memoized.

Fig. 2 shows the percent of dynamic instructions looked up (out of **all** executed instructions, including branches) and the fraction of successful lookups for all 26 applications in the SPEC CPU2000 benchmark. Nearly 75% of all dynamic instructions can be successfully memoized for the CINT2000 suite and 65% for the CFP2000 suite. The weighted harmonic mean is used for all averages in this study. It was chosen for its mathematical properties that are best suited for rates. A study by Yi and Lilja [16] using 7 CPU2000 benchmarks and different metrics concludes that the benchmarks have significant amounts of redundant computations. Our study ratifies these conclusions on the whole suite.

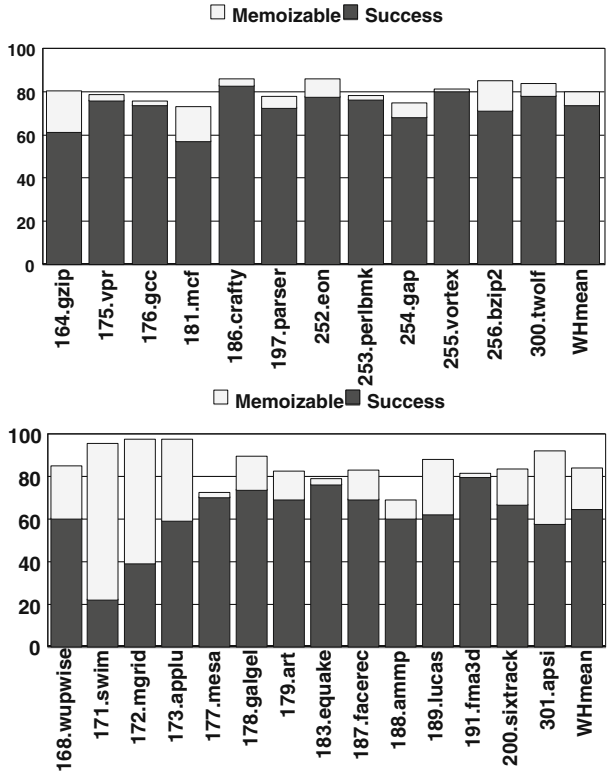


Fig. 2. Percent of all memoizable and successfully memoizable instructions when using a very large MEMO-TABLE

3 MEMO-TABLE Structural Factors

The factors that influence the reuse rate, access time, energy consumption, and area are numerous: size, associativity, indexing, number of ports, etc. A full factorial design would entail hundreds of simulations. In order to reduce this to a manageable size we will first perform a 2^k factorial design using four factors (Sect. 3.1) and then perform a full two-factor design (Sect. 3.2) on the influential factors.

3.1 2^k Factorial Design

A 2^k Factorial Design [17] is used to determine the effect of k factors, each of which has two levels. When a factor has a continuity of levels two extremes are chosen. The technique computes the allocation of variation contributed to each factor separately and in combination with others. The factors and levels we chose are described in Table 3. The metrics measured are:

1. Reuse rates for the CINT and CFP benchmarks measured by dividing the number of successful memoizations by the number of memoizable instructions executed (and then taking the weighted harmonic mean).

Table 3. MEMO-TABLE factors and levels used in the 2^k factorial design

Factor	*Low Level*	*High Level*
size	32 entries	1024 entries
associativity	direct-map	8-way
indexing mode	PC	operands + opcode
replacement mode	random	Least Recently Used

2. Energy of an access (nJ).
3. Access time (ns).
4. Total area (mm^2).

One anomaly that arises is the use of the replacement method as a factor in conjunction with a direct mapped table. We chose to keep this level of associativity due to its influence on the access time and energy results. Access time and energy are calculated using CACTI 3.0 [18] modified to accommodate the large tag size[1] and to distinguish between different indexing and replacement modes. One read port, one write port and a technology of 90nm (forecast for next generation technology) are configured. The allocation of variation is summed in Table 4 and the raw results are in Table 5. From both we can conclude:

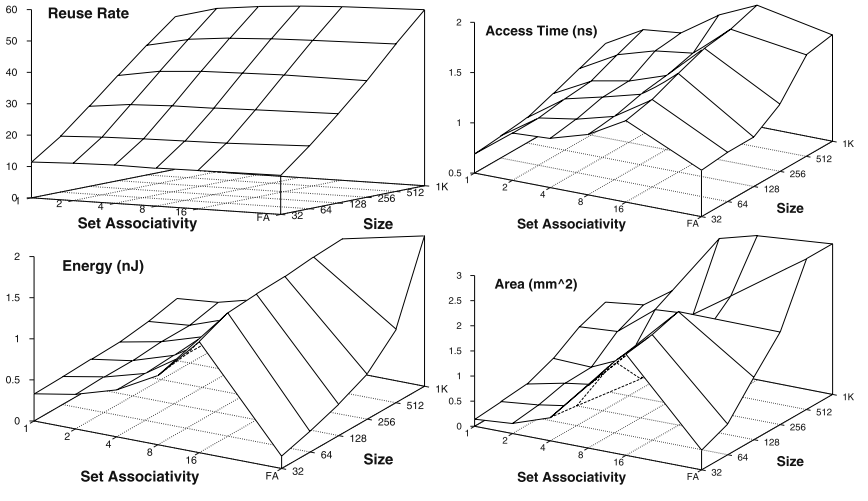


Fig. 3. Reuse rate, access time, energy, and table area as a function of a MEMO-TABLE's size and associativity

1. The size of the MEMO-TABLE is the dominant factor for the reuse rate metric, primary for energy, but is only secondary to associativity for access time and area. Further exploration is needed to determine the optimal table size and associativity.

[1] Two 64-bit operands were simulated, three operand tables are discussed in Sect. 4.1.

Table 4. Allocation of variation of the 2^k factorial design (Size, Associativity, Mapping, Replacement, Correlations between factors)

Metric	*Allocation of Variation (%)*				
	S	A	M	R	C
Reuse Rate (CINT)	87	2	9	0	1
Reuse Rate (CFP)	95	2	2	0	1
Energy	9	90	0	0	1
Access Time	56	41	2	0	1
Area	64	36	0	0	0

Table 5. Configurations and results of 2^k factorial design. S- size, A - associativity, M- mapping, R - replacement method

Configuration	*Results*					*Configuration*	*Results*				
S A M R	cint	cfp	nJ	ns	mm^2	S A M R	cint	cfp	nJ	ns	mm^2
32 1 pc rnd	5.1	7.2	0.34	0.65	0.14	1K 1 pc rnd	36.6	37.0	0.47	1.00	0.97
32 1 pc lru	5.1	7.2	0.34	0.65	0.14	1K 1 pc lru	36.6	37.0	0.47	1.00	0.98
32 1 ops rnd	13.9	11.4	0.35	0.69	0.14	1K 1 ops rnd	55.8	43.5	0.51	1.04	0.97
32 1 ops lru	13.9	11.4	0.35	0.69	0.14	1K 1 ops lru	55.8	43.5	0.51	1.04	0.98
32 8 pc rnd	8.4	9.4	0.57	1.06	1.24	1K 8 pc rnd	47.0	45.0	0.98	1.36	2.04
32 8 pc lru	8.9	9.8	0.57	1.10	1.25	1K 8 pc lru	49.0	47.0	1.01	1.36	2.05
32 8 ops rnd	15.1	13.9	0.58	1.06	1.25	1K 8 ops rnd	64.4	50.4	1.02	1.40	2.04
32 8 ops lru	15.4	14.4	0.58	1.06	1.26	1K 8 ops lru	66.4	52.0	1.04	1.40	2.05

2. The replacement method has hardly any effect on any of the metrics.
3. Using the program counter as the index yields poor reuse rates yet hardly effects time, energy, or area. The reduced reuse rate is a result of: (i) all dynamic instances of an instruction are mapped to the same set, reuse is limited to the size of the set; (ii) dynamic instructions of different static instructions (with the same opcode) can't use each others results. Less than half the successful lookups can be attributed to the same static instruction.
4. When assuming a clock rate of 2GHz (minimum estimate for future IBM POWER implementations) even the fastest configurations take more than one cycle to complete a lookup. This must be: a) minimized. b) compared against the latency of memoized instructions.

After fixing the mapping (operands) and replacement (random) modes, the next step in our study is to perform a full two-factor factorial design using size and associativity.

3.2 Full Two-Factor Factorial Design

In this set of experiments we vary the size of the MEMO-TABLE from 32 to 1024 entries and the degree of associativity from direct-mapped to 16-way and 64-way (which represents fully associative in our model). Indexing is performed using the operands and

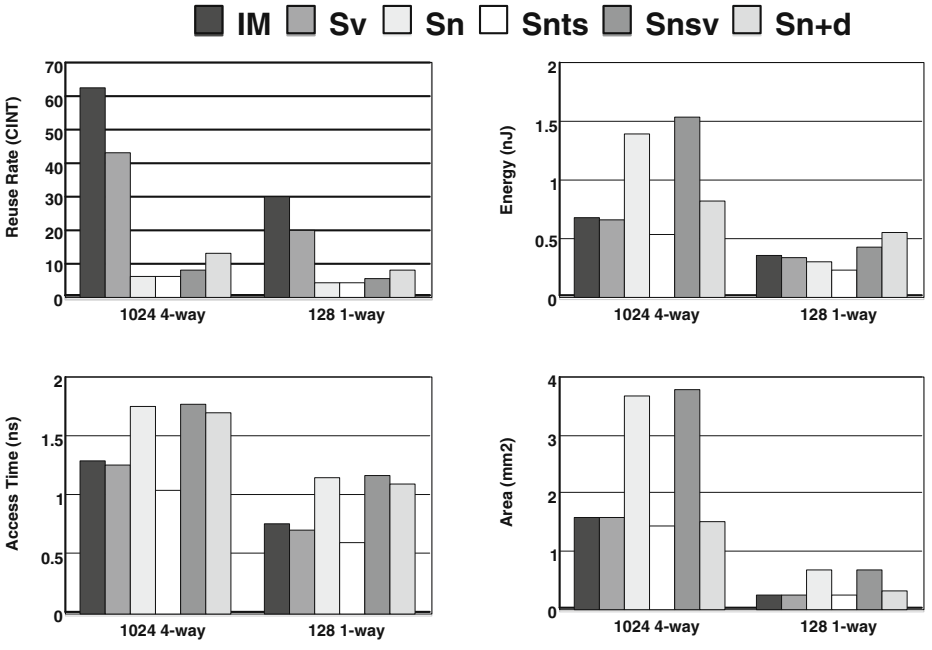


Fig. 4. Reuse rates (CINT), access time, energy, and table area of different memoization and reuse schemes

opcodes and random replacement is implemented. Fig. 3 shows the reuse rates[2], energy, access time, and area (Z-axis) as a function of size (Y-axis) and associativity (X-axis). A surprising result is that small fully-associative tables are faster and consume less energy than corresponding (same number of entries) set-associative tables. This is due to the CAM (Contents Addressable Memory) based design of a fully-associative cache in the CACTI model. For small MEMO-TABLES the overhead of tag decode, routing, and comparison out-weighs the added delay and energy of the large CAM cells (but not the area occupied). However, the almost negligible effect that associativity has on the reuse rate indicates that a direct-mapped MEMO-TABLE is a much better choice.

Assuming a clock rate of 2GHz it would be wise to choose a configuration that minimizes the number of cycles it takes to access the MEMO-TABLE. A 512-entry, direct-mapped MEMO-TABLE has an access time of just under 2-cycles (0.94 ns), a reuse rate of 47.7% (37.9% for CFP), an energy consumption of 0.41 nJoules, and a total area of 0.40 mm^2. In Sect. 4 we will show several techniques to reduce the MEMO-TABLE's size yet retain its reuse rate.

3.3 Instruction Reuse

The large overhead attributed to the tag comparison, the fact that a fully-associative MEMO-TABLE is feasible, and the latency of a MEMO-TABLE lookup, leads us to re-

[2] Just the CINT suite, the CFP suite displays similar behavior.

examine the Instruction Reuse (IR) scheme of Sohi and Sodani (see [7] for full details). IR has three versions:

S_v The PC is used to index the Reuse Buffer (RB), the operand values are used to verify reuse. This is similar to the configurations in Sect. 3.1 where mapping is performed by the PC. The difference is that the PC must match and the opcodes are omitted.

S_n An entry is mapped by the PC and stores only the operand's register names, which reduces the tag size. Every time a register is overwritten the corresponding entries are invalidated. Reuse is verified by a PC match and valid bit. However, the scheme implies a CAM based design necessary for the invalidations.

S_{n+d} In order to overcome frequent invalidations, consuming instructions are linked to their producers. An entry is valid if its producer is in the table and is the last producer of the register value (an auxiliary table maps each architected register to the RB entry which has its latest result). Thus, a lookup can be composed of a RB read and two accesses to the auxiliary table.

Two possible optimizations are using time stamps to test if an operand register was overwritten (this simplifies the RB structure), and not invalidating a register that has been overwritten with its current value (reduces the invalidation rate). Table 6 lists all configurations compared and Fig. 4 displays the results using 1024-entry, 4-way tables and 128-entry, direct-mapped tables. Although the Snts scheme has better physical metrics than IM, and even assuming the reuse rate of the Snsv scheme, the diminished reuse rate makes it unattractive for future microprocessor enhancements.

Table 6. Instruction Reuse Configurations

Name	*Configuration*
IM	MEMO-TABLE described in Sect. 3.2
Sv	S_v scheme described in [7] (similar to MEMO-TABLE mapped by PC)
Sn	S_n scheme of [7], built with CAM cells
Snts	S_n scheme using time stamps
Snsv	S_n scheme, same value doesn't invalidate
Sn+d	S_{n+d} scheme of [7] (incorporates Snts and Snsv techniques)

4 Improving the Reuse Rate

The results obtained in the previous section, 47.7% (CINT) and 37.9% (CFP), are moderate at best. In this section we will suggest several techniques for enhancing the reuse rate and examine their influence on time, energy, and area.

4.1 Multi MEMO-TABLES

In the previous experiments all memoized instructions have been "lumped" together into one table. This is unnecessary and even contradictory to the design of the processor's datapath where instructions are dispatched to different queues and/or reservation stations prior to execution. Using this logic we split the MEMO-TABLE into 3 distinct tables: Integer

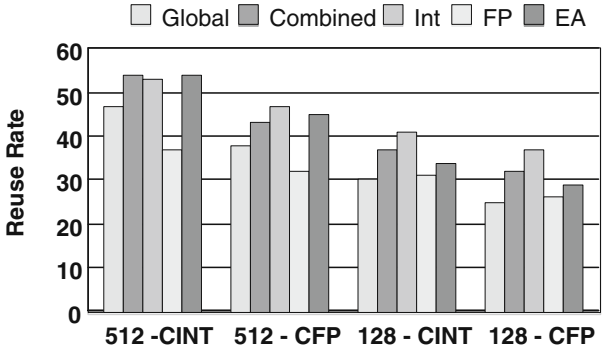


Fig. 5. Reuse rates when the global MEMO-TABLE is split into Integer, Float, and EA calculation tables. MEMO-TABLE sizes are 512 and 128 entries (direct-mapped)

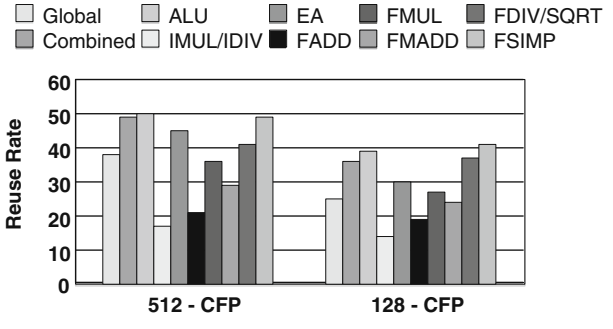


Fig. 6. Reuse rates when the global MEMO-TABLE is split into 8 distinct tables. MEMO-TABLE sizes are 512 and 128 entries (direct-mapped). Only CFP results are shown

operations, FP operations, and Effective Address (EA) calculation. In order to further enhance reuse chances we mapped the FP MEMO-TABLE using a mix of bits from the exponent and mantissa.

The reuse rates for three 512-entry tables and three 128-entry tables are shown in Fig. 5. In addition, the combined reuse rate is shown (number of total successes divided by number of total accesses). From a performance perspective it is clearly beneficial to split the global MEMO-TABLE. The combined, Integer, and EA reuse rates all improve. However, the size of the MEMO-TABLES is now trebled. The reuse rate of three smaller MEMO-TABLES used to approximate one larger MEMO-TABLE falls slightly lower than the monolithic approach. Nevertheless, the size of the three 512 MEMO-TABLES is less than half the size of a 32KB on-chip cache, no extra energy is being expended, and the same number of access is being made. This technique can be further fostered by splitting the Integer MEMO-TABLE into short and long latency instructions (IMUL and IDIV) and by splitting the FP MEMO-TABLE into FADD/FSUB, FMUL, FDIV/FSQRT, FMADD, and all other FP instructions. The rationale is to cluster operations with similar latencies into the same MEMO-TABLE.

Fig. 6 shows the reuse rates for this 8-way split (the majority of the MEMO-TABLES store FP calculations so only the CFP results are shown). The results are mixed, the combined reuse rate is the same as for one 512-entry global MEMO-TABLE, and it surpasses three 128-entry MEMO-TABLES (Fig. 5). However, this is achieved with twice the area. Reasons to implement such a configuration would be for chip locality: moving the MEMO-TABLE closer to the Functional Unit (FU) it serves can reduce wire delay. This also enables building MEMO-TABLES with different characteristics: A SQRT table needs only one operand while a FMADD table needs three.

4.2 Trivial Operations

Trivial operation detection has been used in the past as a memoization filter. Both Richardson [9] and Citron et al. [8] have used it in their works. Operations are defined as trivial when they can be matched against constant values (0,1,-1) and their results are straightforward from the operands ($a + 0 = a$, $a \times 1 = a$, $a/1 = a$, ...). The premise is that instead of storing these operations they will be detected by dedicated circuitry before or in parallel to a MEMO-TABLE lookup. In fact, the trivial operation detection can be viewed as an extra degree of associativity.

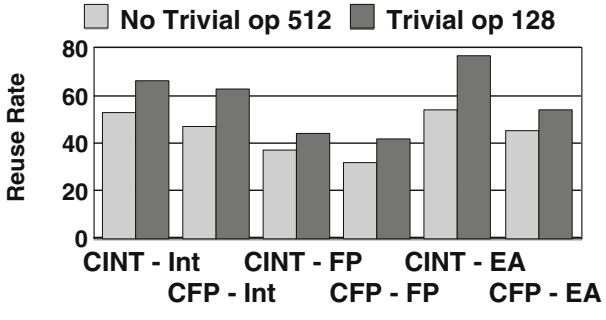


Fig. 7. Comparison of reuse rates of 512-entry MEMO-TABLES to 128-entry MEMO-TABLES with trivial operation detection

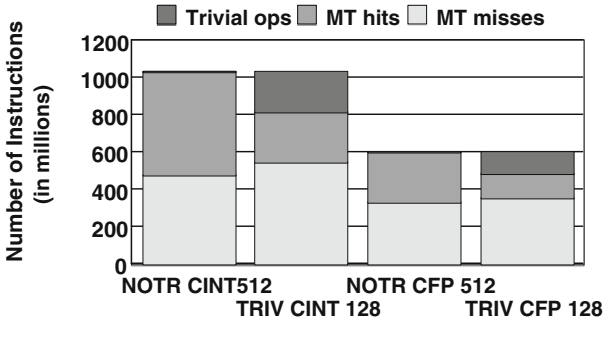


Fig. 8. Comparison of the number of successfully memoized instructions of 512-entry MEMO-TABLES and 128-entry MEMO-TABLES with trivial operation detection

Fig. 7 shows the reuse rate of the three major MEMO-TABLES (Integer, Float, EA) with and without trivial operation detection (in this case a trivial operation detection is considered a successful lookup). At a first glance the results are impressive: for both suites all MEMO-TABLES display an enhanced reuse rate. Moreover, a second, closer, look at the column labels shows that we are comparing 512-entry MEMO-TABLES to 128-entry MEMO-TABLES. By using trivial operation detection we have quartered the size of the MEMO-TABLES, and improved the reuse rate. Fig. 8 compares the raw number of accesses to MEMO-TABLES in both cases (average number of access per benchmark). When using the smaller tables the number of misses is larger: only non-trivial operations are memoized, but there are less accesses which saves energy (in addition to the smaller table sizes).

Nevertheless, trivial operation detection isn't free. Our calculations show that a 0,1,-1 detector for two 64-bit operands has an energy consumption of 0.00051nJ and an area of $0.0023mm^2$. These are inconsequential when compared to the 0.35nJ and $0.24mm^2$ of a direct-mapped, 128-entry MEMO-TABLE. However, it has an access time of 0.21ns. Accessing it in parallel to the MEMO-TABLE hides this latency yet burns energy. A sequential lookup (first trivial operation then MEMO-TABLE) results in an access time of 0.96ns (0.21 + 0.75) which is comparable to a 512-entry, direct-mapped MEMO-TABLE (0.94ns) and is just under two clock cycles for a 2GHz clock. Yi and Lilja [19] suggest detecting trivial operations earlier in the pipeline by testing the first operand to arrive, this can solve the delay problem and should be considered in a detailed pipeline model.

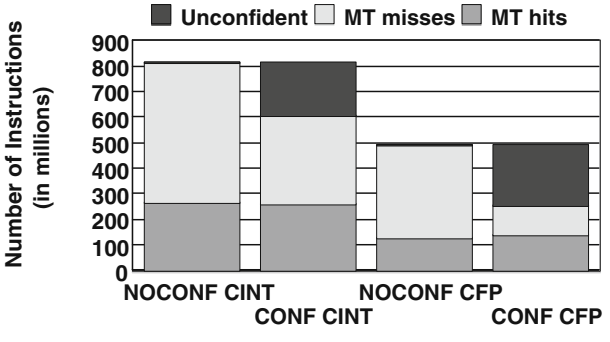


Fig. 9. Comparison of the number of successfully memoized instructions of 128-entry MEMO-TABLES, with and without confidence filtering. Trivial operations aren't counted

4.3 Confidence Counters

Although trivial operation detection reduces the number of MEMO-TABLE accesses the number of misses is still high. Some benchmarks just aren't amenable to memoization (`171.swim` and `171.mgrid` for instance). A well known technique for filtering these wasteful MEMO-TABLE accesses is the use of *confidence counters*. They are usually used for branch [20] and value prediction [21].

In our model every instruction fetched is mapped to a Confidence Table (CT) which contains a n-bit saturating counter per entry. When a dynamic instruction instance hits

the counter is decreased, when it misses it is increased. After n misses a static instruction is marked as non-memoizable (although it can still be tested for triviality) and this data is passed on with it to the MEMO-TABLES. After a predetermined number of cycles the CT is flushed in order to give "mis-memoizing" instructions a second chance.

Fig. 9 displays the numbers of MEMO-TABLE hits and misses when a 5-bit saturating confidence counter is used per entry (trivial ops aren't counted so we can directly compare MEMO-TABLE misses). The CT contains 1024 entries and is flushed every 131072 cycles and the MEMO-TABLES are direct-mapped with 128 entries and trivial op detection. The results are impressive, the CT manages to filter out many unsuccessful instructions while raising the number the memoized instructions. The "price" is a table that has an energy consumption of 0.011nJ, an area of $0.087mm^2$, and an access time of 0.24ns. The energy savings are huge, every aborted lookup saves $0.35 - 0.011 = 0.339nJ$ and the CT can be accessed way before memoization, hiding the CT's latency. The only complexity is linking the results of the MEMO-TABLE lookups to the CT, this has to be explored in a detailed datapath design. The CT can be reduced even further to 256 entries and 4 bits per counter, and still retain a better reuse rate than not using confidence counters. This shaves off several picoseconds from the access time ($0.24ns - 0.21ns = 30ps$).

Table 7. Characteristics of 64-bit functional units and their adjacent MEMO-TABLES

Functional Unit	*Unit Features*			*CFP Rates*		
	latency	*energy*	*area*	*confidence*	*reuse*	*trivial*
IADD	1	0.16	0.09	28.4	21.4	29.6
IMUL	7	1.97	0.20	4.4	3.8	9.2
IDIV	68	3.63	0.20	19.7	14.8	75.7
FADD	6	0.69	0.48	21.5	14.2	33.4
FMUL	6	1.17	0.52	14.4	6.4	32.9
FMADD	6	1.37	0.72	17.7	6.9	19.4
FDIV	30	5.51	0.48	12.9	8.5	38.8
MEMO-TABLE	2	0.35	0.24	128-entry, 1-way; update energy: 0.17nJ		
TO detector	1	0.00051	0.0023	detects 0,1,-1 patterns		
CT	0	0.011	0.087	1024-entry, 5-bit saturating counter		

5 "Look It Up" or "Do the Math"?

Finally after exploring the range of MEMO-TABLE attributes we must compare the memoization paradigm to the basic computations themselves. Table 7 lists the characteristics of several 64-bit functional units and the reuse[3] (rr), trivial operation (tr), and confidence (cr) rates (instructions that haven't failed memoization) of the 128-entry MEMO-TABLES servicing them. The energy data was obtained from the PowerTimer [22] tool which models a 64-bit processor [4]. It is assumed that the MEMO-TABLE lookup is performed in parallel to computation and squashes it upon success. All instructions access the CT

[3] The number of successful memoizations divided by the **total** number of instructions executed.
[4] This isn't official IBM data and shouldn't be pertained as such.

and memoized instructions update it as well. To measure the usefulness of memoization we defined two equations not unlike the Average Memory Access Time (AMAT).

ACT ***A**verage **C**omputation **T**ime* The average time (in cycles) to compute an operation: $ACT = rrMT_t + trTO_t + [1 - (rr + tr)]FU_t$

ACE ***A**verage **C**omputation **E**nergy* The average energy (nJoules) expended when computing an operation. This is slightly more complex and takes into consideration the MEMO-TABLE lookup ($MT_{lk.e}$: computed for all non-trivial operations that passed the confidence test), update energies ($MT_{up.e}$: computed for all MEMO-TABLE misses), and the energy of the FU until the operation is squashed ($FU_{sq.e}$: computed for all MEMO-TABLE hits). The TO detector and CT are accessed by all instructions, the CT is updated by all memoizable instructions: $ACE = TO_e + (1 + cr)CT_e + crMT_{lk.e} + rrFU_{sq.e} + (cr - rr)MT_{up.e} + [1 - (tr + rr)]FU_e$

Fig. 10 shows the ACT and ACE of the afore listed units compared to the latencies and energies without memoization (the CFP2000 suite is used). The ACT and ACE both show that it is counterproductive to memoize integer addition instructions, it incurs both performance and energy penalties. All other units display performance and energy gains. The gains are proportional to the units latency, the longer the latency the higher

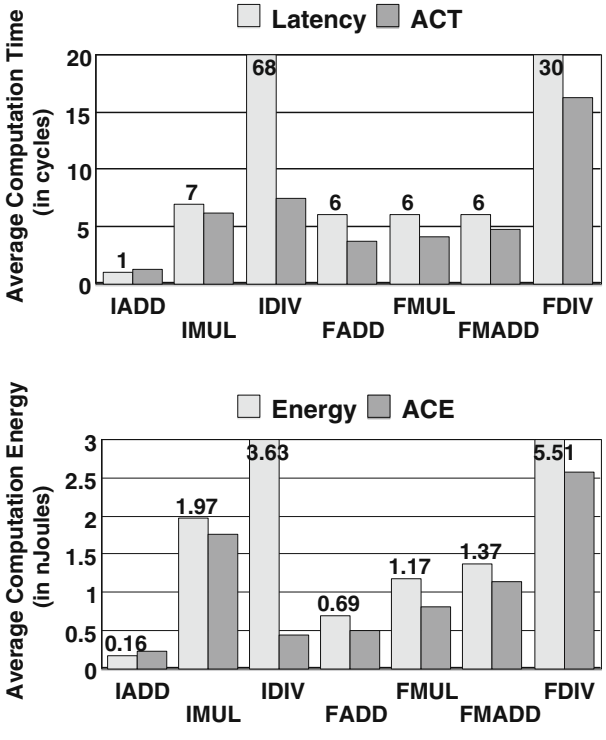


Fig. 10. Average Computation Time (ACT) and Average Computation Energy (ACE) compared to the latencies and energy consumption of seven functional units. The results are for CFP2000

the performance potential. The problem is that long latency instructions usually have a low frequency of execution. This must be overcome in future work.

6 Observations and Conclusions

Several key observations have been noticed during this study and will define the basis for future instruction memoization exploration:

- Reuse opportunities are rampant in SPEC CPU2000. 65-75% of all dynamic instructions have been executed with the same operands previously.
- Mapping the MEMO-TABLES using the operand values utilizes the full table and enables dynamic instruction instances of different static instructions to use each others results.
- The associativity of a MEMO-TABLE profoundly affects its access time, energy, and size yet hardly enhances its reuse rate. Direct-mapped is the way to go.
- Directing instructions to several MEMO-TABLES based on instruction classes is more cost effective than a single monolithic table.
- Trivial operation detection can quarter a MEMO-TABLE's size while increasing its reuse rate.
- Confidence counters filter out many un-memoizable instructions without reducing the number of successful MEMO-TABLE lookups.
- A comparison between direct computation to computation + memoization shows that it is useless to memoize single-cycle instructions.
- Memoization of long latency instructions shows a potential for performance improvement, and due to the use of confidence counters memoization results in energy savings for most units.

This study is the first step in proving that instruction memoization is a viable performance improving technique for modern microprocessors. We have shown that it is possible to obtain high reuse rates combined with low energy penalties and area overhead. Nonetheless, there is still plenty of work ahead: in light of our observations we must now integrate IM into a detailed pipeline model. The stages in which to perform confidence and trivial operation tests must be chosen, operands must be supplied to the MEMO-TABLES as early as possible, and the effects of compiler scheduling must be examined. In addition integration of dependent instructions into one memoization unit should be further explored (similar to the S_{n+d} scheme and the Dynamic Computation Reuse scheme of Connors and Hwu [23]).

The bottom line: Fast clock rates are increasing the latency of many complex instructions. Instruction Memoization can reduce these latencies and reduce energy consumption to boot.

References

1. O'Connell, F., White, S.: POWER3: the next generation of PowerPC processors. IBM Journal of Research and Development **44** (2000) 873–884

2. Vetter, S., et al.: The POWER4 Processor Introduction and Tuning Guide. IBM. (2001)
3. Intel Corporation: (Differences in Optimizing for the Pentium 4 Processor vs. the Pentium III Processor)
4. Intel Corporation: IA-32 Intel® Architecture Optimization Reference Manual. (2003)
5. (http://www.sun.com/processors/UltraSPARC-II/details.html)
6. Sun Microsystems: UltraSPARC III User Manual. 2.2 edn. (2003)
7. Sodani, A., Sohi, G.: Dynamic Instruction Reuse. In: Proceedings of the 24th International Symposium on Computer Architecture. (1997)
8. Citron, D., Feitelson, D., Rudolph, L.: Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units. In: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operationg Systems. (1998) 252–261
9. Richardson, S.: Exploiting Trivial and Redundant Computation. In: Proceedings of the 11th Symposium on Computer Arithmetic. (1993) 220–227
10. Molina, C., González, A., Tubella, J.: Dynamic Removal of Redundant Computations. In: Proceedings of the 1999 International Conference on Supercomputing. (1999) 474–481
11. Azam, M., Franzon, P., Liu, W.: Low Power Data Processing by Elimination of Redundant Computations. In: Proceedings of the 7th International Symposium on Low Power Electronics and Design. (1997) 259–264
12. Citron, D., Feitelson, D.: Revisiting Instruction Level Reuse. In: Proceedings of the 1st Workshop on Duplicating, Deconstructing, and Debunking. (2002) 62–70
13. Tendler, J.M., Dodson, J.S., J. S. Fields, J., Le, H., Sinharoy, B.: POWER4 system microarchitecture. IBM Journal of Research and Development **46** (2002) 5–26
14. Moudgill, M., Wellman, J., Moreno, J.: Environment for PowerPC Microarchitecture Exploration. IEEE Micro **19** (1999) 15–25
15. KleinOsowski, A., Lilja, D.J.: MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. Computer Architecture Letters **1** (2002)
16. Yi, J., Lilja, D.: An Analysis of the Amount of Global Level Redundant Computation in the SPEC 95 and SPEC 2000 Benchmarks. In: Proceedings of the 4th Annual Workshop on Workload Characterization. (2001)
17. Jain, R.: The Art of Computer Systems Performance Analysis. Wiley Professional Computing (1992)
18. Shivakumar, P., Jouppi, N.: CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical report, Compaq: Western Research Laboratory (2001)
19. Yi, J., Lilja, D.: Improving Processor Performance by Simplifying and Bypassing Trivial Computations. In: Proceedings of the 20th International Conference on Computer Design. (2002)
20. Jacobsen, E., Rotenberg, E., Smith, J.: Assigning Confidence to Conditional Branch Predictions. In: Proceedings of the 29th International Symposium on Microarchitecture. (1996) 142–152
21. Burtscher, M., Zorn, B.G.: Prediction Outcome History-based Confidence Estimation for Load Value Prediction. Journal of Instruction-Level Parallelism **1** (1999)
22. Brooks, D., Bose, P., Srinivasan, V., Gschwind, M.K., Emma, P.G., Rosenfield, M.G.: New methodology for early-stage microarchitecture-level power-performance analysis of microprocessors. IBM Journal of Research and Development **47** (2003) 653–670
23. Connors, D., mei Hwu, W.: Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In: Proceedings of the 32nd International Symposium on Microarchitecture. (1999) 158–169

CPU Packing for Multiprocessor Power Reduction

Soraya Ghiasi[1] and Wes Felter[2]

[1] University of Colorado
ghiasi@cs.colorado.edu
[2] IBM Austin Research Laboratory
wmf@us.ibm.com

Abstract. Power and cooling considerations have moved to the forefront of modern system design. The restrictions placed upon systems by power and cooling requirements have focused much research on a variety of techniques to reduce maximum power and leakage. Simultaneously, efforts are being made to adapt microarchitectural features to the current needs of an application. We focus instead on adapting large scale resources to the current needs of a server farm.

We study the efficacy of powering on and off CPUs in symmetric multiprocessors (SMP). We develop a number of different predictive and reactive techniques for identifying when cores should have their state altered. We present results for these policies and find a hybrid policy presents a reasonable balance between the time necessary to predict future needs and the accuracy of these predictions. It maintains 97% of the original system performance while reducing the energy per web interaction by 25%.

1 Introduction

Power has become a major design consideration in modern processors. The limited battery capacity of portable devices has led to the inclusion of a variety of power-saving techniques including voltage scaling and clock gating. While early efforts have focused on mobile devices, power and thermal concerns are becoming problematic even in servers. Current symmetric multiprocessor (SMP) servers have few, if any, CPU power-saving features beyond clock gating. In many servers, the CPUs consume a substantial fraction of the system power [1].

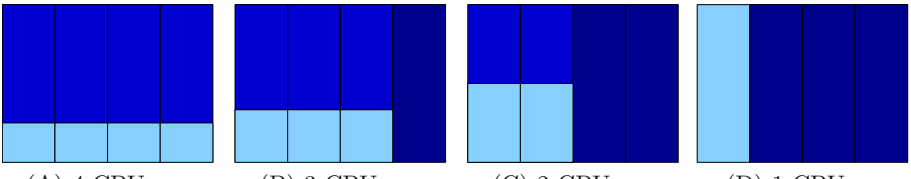
Many servers have variable workloads and thus variable resource utilization. Figure 4 shows the number of connections made to a Web server over the course of a 24 hour period. If these requests are distributed across a 4-way SMP, there are periods during which the demand can be handled by fewer than 4 CPUs. These periods of low utilization can be exploited to reduce the operating power of a server.

This paper describes a method for integrating SMP CPU power management with the CPU scheduler in the Linux 2.5 kernel. When a system is completely

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 117–131, 2004.

utilized, there is no opportunity for saving CPU power without reducing performance; all CPUs must be in the running state. When the workload does not demand full utilization, the default Linux scheduling policy saves no power because the utilization is equalized across all CPUs.

We propose alternate policies which consolidate threads on to the fewest number of CPUs while limiting performance degradation. Because CPUs are not always completely utilized, power can be saved by scaling or turning off unused CPUs. We expect that when the global CPU utilization is less than $(n-m)/n$, then m CPUs out of n can be turned off, saving substantial power. Since CPUs cannot be turned on instantly, some CPU capacity must be held in reserve for bursts. For this reason we never turn off all CPUs.



(A) 4 CPUs on 25% utilization each | (B) 3 CPUs on 33% utilization each | (C) 2 CPUs on 50% utilization each | (D) 1 CPU on 100% utilization

Fig. 1. Different utilization distributions in a 4-way SMP system

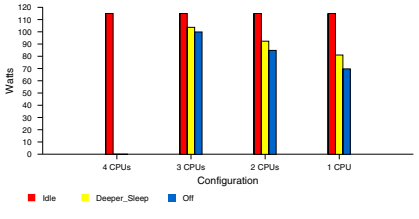
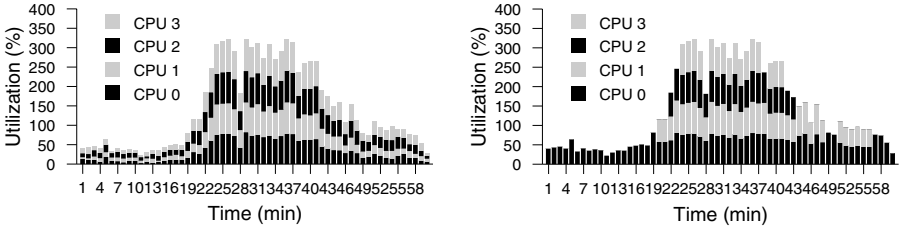


Fig. 2. Applying low power modes, such as deep sleep, to idle processors can significantly reduce the power consumption of the machine. In this example, a total cpu utilization of 100% uses different amounts of energy depending upon the utilization distribution and the low power modes

A simple, illustrative example of CPU packing is presented in Figure 1. A 4-way SMP system has a maximum utilization of 400% (100% * 4 CPUs). In this example, the system has a total utilization of 100%. There are 4 different ways to distribute the workload across the available CPUs. The standard Linux scheduler will load balance and achieve approximately the distribution shown in Figure 1A. Each CPU will carry 25% of the load. However, this is wasteful because 75% of the resources on each CPU are unused. If one CPU was shut down or placed into a deep sleep mode, each of the remaining CPUs would have a utilization of approximately 33% as shown in Figure 1B. Turning off an additional CPU will distribute 50% of the load onto each of the remaining CPUs (Figure 1C). Finally, Figure 1D shows all of the load could be serviced by a single CPU remaining on with a utilization of 100%. Figure 2 shows an estimate of the potential power savings for each of these configurations. The ability to turn CPUs of or even place them into a deep sleep state provides a significant power reduction. Simply idling unused resources does not provide a reduction in power.



(A) Without CPU Packing (B) With CPU Packing

Fig. 3. CPU utilization on a 4-way SMP for a TPC-W workload with the financial profile

A more realistic example is drawn from the workloads we use in this study. The details of the workloads and power metrics used are presented in Section 4. The utilization shown in Figure 3A corresponds to the a normal Linux scheduler running on a 4-way SMP system for the input requests shown in Figure 4A. If we instead take a conservative estimate of allowing no single CPU to reach over 85% CPU utilization while simultaneously packing the available jobs into the minimal number of CPUs, we are able to place unused CPUs into a deep sleep mode or even turn them off entirely. A packed utilization is shown in Figure 3B. Any time a CPU is unused, it can be placed into a lower power mode. When no power savings mode is used, 100% of the original power is used; a deep sleep mode consumes only 55% of the original power; turning idle CPUs off consumes only 40% of the original power.

We will discuss and compare the power and performance of several policies. Although the necessary microarchitectural support is not yet available, a preliminary analysis is possible. In Section 2, we discuss the related work. Section 4 describes our experimental setup and the modifications made to the Linux kernel. Our CPU packing policies are presented in Section 5. Results are discussed in Section 6. Section 7 summarizes our work and presents possible future directions.

2 Related Work

Prior work in the area of OS-level power management has been on uniprocessor machines which use voltage scaling to exploit the energy savings predicted by $P = CV^2f$. Non-uniprocessor efforts have been in the area of cluster systems where it is possible to exploit both voltage scaling and the power reductions via turning on and off machines in the cluster.

Weiser et al. laid the early groundwork for the use of voltage scaling in uniprocessors [2]. Their work introduced the use of the PAST policy which we revisit in this work in the context of SMPs. Govil et al. addressed some of the shortcomings, including jerky responses, of the PAST policy and developed the PEAK

policy in response to the identified problems [3]. Pering et al. investigated the use of interval-based voltage scaling [4, 5]. Their work includes the idea that some intervals can be stretched to meet a deadline rather than completing the work earlier.

Grunwald et al. implemented many of the proposed uniprocessor policies that had previously been studied with simulators[6]. They found that the techniques were often infeasible. In addition, the implemented policies failed to perform as prior simulations had predicted they would.

Flautner et al. studied the effect of automatic performance setting techniques using dynamic voltage scaling on interactive user tasks[7]. Their work eliminated the need for the use of explicit deadlines by deriving deadlines based on kernel activity. This allows voltage scaling techniques to be applied to a wide range of non-periodic and even interactive tasks.

Bohrer et al. were among the first to consider power management for web server farms[8]. They highlighted the fact that the resource requirements of web servers vary over time and that this behavior can be exploited to reduce the overall energy consumption of the server farm. They examined the effect of voltage scaling and frequency scaling on the energy consumption of cluster-based web server farms using a simulator.

Elnozahy et al. studied energy-efficient techniques for managing clusters used in server farms[9]. They examine a number of different policies which use different types of voltage scaling and powering down unused servers in the cluster. They find that using coordinated voltage scaling in conjunction with varying nodes on and off provides the largest energy reductions.

Rajamani and Lefurgy evaluated a variety of power aware request distribution schemes for reducing energy consumption in server clusters used for serving web traffic[10]. Their clusters consist of independent machines each of which can be powered on and off separately. System utilization and system response time are the among the factors considered in their work. Our policies are able to adapt to changing utilizations much more quickly, which allows us to save more power and waste fewer resources than cluster based schemes. Our policies eliminate the need to have spare resources to meet possible upcoming demand.

Our work differs from the prior work in this area by focusing on SMP systems rather than clusters or uniprocessors. We study the effect of using simple uniprocessor-like policies coupled with techniques that are infeasible in uniprocessors. Similarly our work differs from prior cluster-based efforts in that we are able to respond much more rapidly to changes in resource requirements.

3 Model

Go reread the pard-dvs model development to see how they presented it.

This model can be extended to cover NUMA architectures be weighting the power up and down costs by the.

4 Methodology

Although we developed a theoretical model to describe the behavior of a CPU packing-enabled system, we felt that experiments on real, commercially available hardware would be best able to demonstrate the benefit of CPU packing.

4.1 Workload

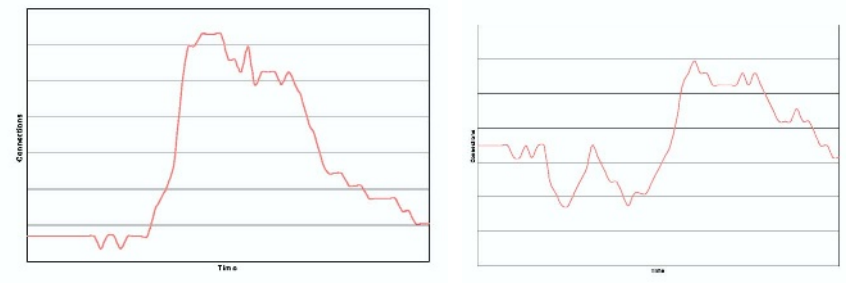
All experiments were performed using the TPC-W benchmark [11, 12]. TPC-W is a transactional Web e-commerce benchmark which simulates an online bookstore. We use a TPC-W implementation based on Apache 1.3.23, PHP 4.1.2, and MySql 4.0.13. We ran the Web server and database on the same machine. Although all three Web interaction mixes were tested initially, we focused our further efforts on WIPS, the Web Interactions Per Second for the shopping mix consisting of 80% browsing and 20% ordering. Preliminary results for WIPSb (95% browsing, 5% ordering) and WIPSo (50% browsing, 50% ordering) modes were also examined, but detailed results are presented only for WIPS.

Because TPC-W is a probabilistic benchmark, our client browser emulator is execution-driven instead of trace-driven. This makes it much easier to change the parameters of the workload, but benchmark runs are not completely repeatable since random numbers are used for think times and state transitions. We modified the client browser emulator to always use the same seed for its random number generator, producing more repeatable runs. We observed a variance of approximately 7% between results of different runs with identical parameters. This variance may be due to interference from other traffic on our test network. Baseline cases, in which no cpu packing was performed, were run five times each. Due to limited resources and the size of the design space we were exploring, we were unable to repeat all subsequent experiments. In light of these constraints, we do not present variances on our individual results, but ask the reader to keep in mind our initial analysis of a 7% variance due to factors not controlled in our experiments.

Our benchmark database consisted of 10,000 items and 288,000 initial customers. The client browser emulator makes a specified number of simultaneous connections to the TPC-W server. We determined that our test system can handle 100 simultaneous connections while meeting all of TPC-W's response time requirements.

Profile Data Based Workloads. Profiled workloads were used to generate variable loads on our server. They specify the number of connections the client browser emulator makes to the TPC-W server. We used two different sets of profile data in conjunction with our client emulator. Both profiles cover 24 hours. The profiles were derived from Web server access logs and have been translated into a format supported by the Rice TPC-W implementation. The first profile is taken from the access logs of a major financial services Web site on October 19, 1999 (see Figure 4A). The second profile derives from access logs collected on February 19, 1998 at the 1998 Nagano Winter Olympics (see Figure 4B).

The profiles are scaled to provide 100 simultaneous connections to our server during their peak utilization. Although these profiles each represent 24 hours, we compressed each profile to a single hour to allow for feasible experimental runs. The shapes of the profiles remain the same, but the rate of change of the number of connections is greater in the compressed profile. Rapid changes in the number of connections and, hence, the utilization often prove difficult to manage, but they are a good test of the applicability of a responsive technique to time-varying demands.



(A) Financial (B) Olympics

Fig. 4. One-day workload profiles used in our experiments

4.2 Hardware

Our experiments were performed on a 4-processor 400 MHz Pentium II Xeon system with a 450NX chipset and 1GB of memory. Each CPU had a 1 MB L2 cache.

4.3 Prototype Implementation

We conducted our experiments using Linux 2.5.70. Our CPU packing policies are implemented in a userspace application, written in C, which periodically samples the utilization of each CPU using /proc, predicts the aggregate CPU utilization over the next interval, and turns CPUs on or off as necessary.

We are not aware of any SMP servers which allow CPUs to be powered on and off dynamically, so we attempted to simulate this behavior on a normal server. A "CPU Hotplug" patch which allows CPUs to be enabled and disabled dynamically is in development for Linux[13], but it was not complete enough for our experiments. With these restrictions in mind, we used a simple user space technique to simulate turning CPUs on and off.

To simulate a CPU that is off, we create a process with SCHED_FIFO scheduling policy and set its affinity for the CPU that is to be turned off; this process simply clears the caches and then busy-waits until it receives a signal. Since SCHED_FIFO processes owned by the root user are never preempted and have strictly higher priority than normal (SCHED_OTHER) processes, the Linux scheduler migrates all other processes off the target CPU. When this realtime

process is running, the CPU is considered "off". When the workload demands that the CPU be reclaimed, we send a signal to kill the "off" process. This approach allowed us to simulate CPU packing with no changes to the kernel.

The policies considered are explored in depth in Section 5.

4.4 Power Estimation

Since we could not actually turn off CPUs during our experiments, we cannot measure the power savings caused by CPU packing by directly measuring CPU power consumption. However we are able to estimate CPU power over a run based on a CPU utilization trace because power consumption of modern CPUs is highly correlated to utilization. Our CPU packer writes its power management decisions to a log, which allows us to properly account for power using postprocessing.

Our power model is parameterized using measurements of a 3.06GHz Pentium 4 (Northwood) with HyperThreading disabled. We use data from a more modern processor because we feel these are more representative of the current state of low power design; specifically, we were interested in using numbers that reflect the effect of clock gating. We focus only on the processor power, but it is possible that relative power savings will be higher than those reported in this study because of the ability to turn off fans as well.

We use the least squares fit of the measured data to determine how much power is being consumed by each processor in the SMP based on its current utilization. $Power = 0.547WU_t + 15.064W$ is used throughout our calculations. The total power consumption is simply the sum of individual power consumptions for each CPU. In this paper, we present only power consumed by CPUs; a complete system will consume more power.

4.5 Metrics

Our primary performance metric is whether a run meets the response time criteria defined by TPC-W; these criteria require that 90% of the interactions of each type have a response time less than a defined threshold, which varies for each interaction type. These response time thresholds are in the range of 3-20 seconds. The response time metric does not represent a complete picture of how a policy performs.

We are also interested in system throughput which we define as the number of completed interactions during a run. Although the number of simultaneous connections is fixed by the profile, the number of interactions completed by each connection depends on the response time of each interaction, which depends on system utilization. Thus we observed varying throughput even though the load and response time requirements were the same for all of our experiments.

Because load varies during our experiments, power consumption also varies. The total energy consumed during a run is not a good metric since the number of interactions varies. Thus we focus on energy per interaction as our primary metric of power savings.

We are also interested in the number of transitions made between states. A high number of transitions can be indicative of responsiveness to change or an indication that the current data sampling window is too small. In addition, it is currently unclear what impact, if any, a high number of transitions between the on and off states may have on the reliability and failure rates of CPUs. Frequent cycling may introduce stresses which can reduce the reliability of CPUs. We choose a conservative approach if it does not have a signficantly negative impact on the other metrics.

4.6 Assumptions

We make a number of conservative assumptions in our experiments. We intentionally overestimate the time and power penalties associated with transitioning from "on" to "off" and "off" to "on". In addition, we overestimate the power consumed by the "off" state. These assumptions are itemized below.

- **Utilization Thresholds -** We increase the number of CPUs, m, by one when the utilization is greater than 85% of the current supportable maximum. We decrease the processors by when the utilization is less than 85% of the maximum total utilization of $(m-1)$ CPUs.
- **Transition Times -** Our original intent was to use the *CPU hotplug* patch to the Linux kernel to actually remove processors from the pool of processors available for scheduling. However, at the time of our experiments, this patch did not work on our 4-way server. We measured the transition time to be $<<$ 0.5 seconds on a different system. We conservatively use 0.5 seconds as our transition time for turning processors"on" and "off". During this time, no processes may be scheduled onto the transitioning processor.
- **Transition Power Costs -** We chose to use a real time process to mimic removing a processor from the pool, but our CPU packer now must be more aware of the transitioning of states so that we can correctly record what would have happened if the processor had actually been turned off. We conservatively assume that the processor is fully utilized during both transitions. In reality, the transition process is not CPU intensive, but we chose to model it as such to ensure that we do not underestimate the power costs of transitions.
- **Power Consumption in the Off State -** We assume that the CPU is placed in a deep sleep state similar to Intel's "Deeper Sleep" mode. Rather than consume zero watts, we assume that our "off" state consumes half as much power as a running processor with zero utilization. Such a state does not currently exist on Intel's non-mobile processors, but we chose to include the impact of its use in this study. In mobile processors, the "Deeper Sleep" mode consumes less power than the "Deep Sleep" mode by scaling Vdd in addition to gating the chip input clock.

5 CPU Packing Policies

A CPU packing policy monitors the CPU utilization and determines if and when a CPU should change state. When the aggregate CPU utilization drops below the utilization that can be supported by N-1 CPUs, CPU N is turned "off". Similarly, when the utilization criteria indicated that the N CPUs are overly utilized and there are CPUs in the "off" state, CPU N+1 is turned "on". The process of turning "on" and "off" CPUs involves transitioning through an additional state in each direction. These states, "up" and "down", are necessary because they transition architectural and cache data to and from the CPU. We assumed that "up" and "down" each take 0.5 seconds and consume full CPU power during the transition. This simple mechanism is then used in the context of our different CPU packing policies.

```
4 //Number of CPUs
3 //First CPU to turn off
2 //Second CPU to turn off
1 //Third CPU to turn off
0 //Always on CPU
```

Fig. 5. Relationship File specifies how many CPUs followed by the order in which they are turned off

We examine a number of existing policies. These policies have in the past been applied to scheduling problems and have been used to control frequency and voltage scaling as well as microarchitectural changes. To our knowledge they have not been applied to the process of predicting the number of processors to be used by an SMP. We chose to examine policies in order of increasing implementation complexity. As a result, we studied no predictive mechanisms because our reactive mechanisms proved adequate. We do include an **oracle** policy to provide an indication of the best possible results using our current set of assumptions (discussed in 4.6).

All our policies use a simple file-based relationship map to determine which processors should be turned on or off. This map can easily be extended to cover much more complicated architectures such as an IBM pSeries 670 which contain multi-chip modules or a Pentium 4 Xeon SMP by taking into account the system design when building the map. Our system consisted of four unrelated processors so we simply shut them off from highest processor number to lowest processor number. We rely on the Linux scheduler to load balance between the "on" processors. The relationship file used in our experiments is shown in Figure 5.

5.1 Oracle

We use the **oracle** policy to provide an upper bound on the achievable results. The data collection run is done in which no CPU packing occurs, but all CPU utilization is collected. The utilization logs are fed to the **oracle** which then determines how many CPUs will be needed at each instant in the run.

5.2 PAST

The PAST policy was originally developed by Weiser et al.[2] for use in uniprocessor systems. PAST uses only utilization information from the past sampling interval and predicts that utilization in the next interval will be the same as the utilization in the previous interval.

5.3 AVG$< N >$

Prior work by Grunwald et al. studied the use of an exponentially decaying average as a utilization predictor[6]. The predicted utilization is governed by equation $W_t = NW_{t-1} + U_t/N + 1$. where W_t is the predicted utilization, W_{t-1} is the prediction from the previous increment, U_t is the current actual utilization, and N is the weighting factor. A small value for N will rapidly reduce the contribution to a negligible contributor while larger values of N include more time. The sampling interval used is 1 second. For this policy, the utilization in the next interval is assumed to be W_t. When this changes enough to indicate a change is necessary, we add or subtract a CPU in our system.

5.4 Hybrid

While working with PAST and AVG$< N >$, we realized that each policy had strengths and weaknesses. We introduce a hybrid policy composed of both PAST and AVG$< N >$ to address some of the shortcomings we observed. The hybrid policy blends the rapid responsiveness of PAST with the longer view of changing demands provided by AVG$< N >$. PAST is used to respond to increasing demands, while AVG$< N >$ is used to decide when CPUs should be turned off. AVG$< N >$ in the hybrid is forced to respond more slowly to reductions in demand by resetting the value of W_{t-1} to U_t whenever PAST turns on another CPU. In our hybrid implementation, a prediction of increased utilization takes precedence over a prediction of decreased utilization.

6 Results

Each of the policies we examine work at least moderately well in same configuration for our workloads. Previous researchers, such as Grunwald, et al [6], found these policies failed when applied to uniprocessor voltage scaling. They work in a CPU Packing SMP because there is usual some amount of unused resources available to meet a sudden increase in demand and our workload demands vary much more slowly than some of the traditional uniprocessor workloads such as mpeg decoding.

6.1 Oracle

The oracle provides an estimate of the maximum possible savings as well as the penalty for guessing wrong. Oracle results are not discussed here, but are used

as a baseline for identifying how well PAST and AVG$< N >$ do in comparison to a reasonable approximation of the best case.

6.2 PAST

Sampling Interval Sensitivity. For the PAST policy, we began by studying the effect of different sampling intervals on the responsiveness of the system. The limiting factor in deciding how many connections to support during peak utilization was the BestSellers Web interaction response time (WIRT) so we focused our attention on the 90th percentile response time for this metric. Figure 6 shows that for both the financial and Olympics data sets, a very short sampling interval is required to meet the BestSellers WIRT criteria. For rapidly changing utilizations, a sampling time of 1 second was found to suffice. Anything longer produced an initial degradation in response time that was unacceptable. This result may be an artifact of scaling 24 hours of profile data into a 1 hour period. Changes between the number of simultaneous connections are dramatic when scaled to such a short time frame, while the changes are more gradual in a longer duration run.

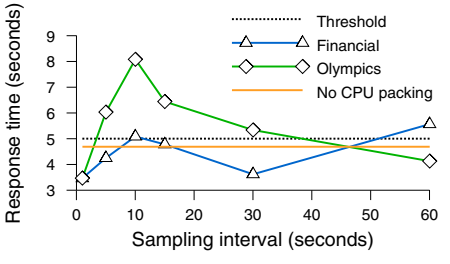


Fig. 6. PAST response time vs. utilization sampling interval

Our second metric of interest was to study the amount of time PAST spends in each of the different system configurations (1 CPU on, 2 CPUs on, 3 CPUs on, 4 CPUs on). A close match to the oracle is desirable because it indicates that the policy was able to correctly respond to changing needs. Figure 7 compares PAST to the perfect Oracle. Financial matches most closely with a sampling interval of 15 seconds, while Olympics matches most closely with a sampling interval of 10 seconds. However, the WIRT discussed above has already limited the solution space to exclude these two sampling intervals.

The effects of different sampling intervals on throughput are shown in Figure 8. As expected from the system configuration data above, financial showed the least throughput degradation (3%) with a sampling window of 15 seconds. Olympics showed a throughput degradation of 2% at a sampling window of 10 seconds. These results indicate that if we could relax the response time criteria, CPU Packing would have a negligible impact on throughput. For TPC-W's response time criteria, however, a 1-second sampling interval causes a 5% throughput degradation. As mentioned previously, our experimental runs were found to have a variance of approximately 7%.

Different Power Modes. Figure 9 shows a projection of the number of joules necessary to complete each Web interaction compared to a system without

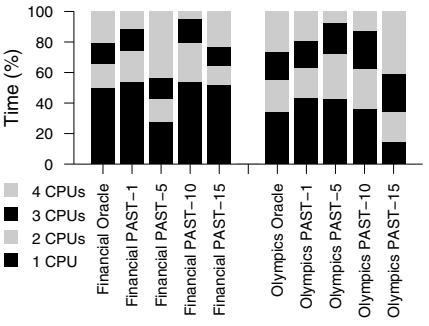


Fig. 7. PAST system configurations vs. utilization sampling interval

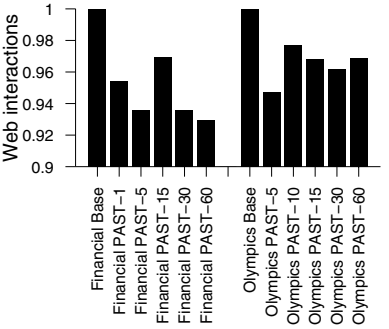


Fig. 8. PAST throughput vs. utilization sampling interval (normalized to base)

CPU packing. The colored bars represent different hardware power modes: none, "Deep Sleep", and off, respectively. Financial with a 5 second sampling interval is a bad case because it consumed nearly as much energy while suffering a 7% throughput degradation. For both financial and Olympics, our chosen 1 second sampling interval shows a significant reduction of energy per interaction. Over 25% less energy is required per interaction when CPU Packing is used.

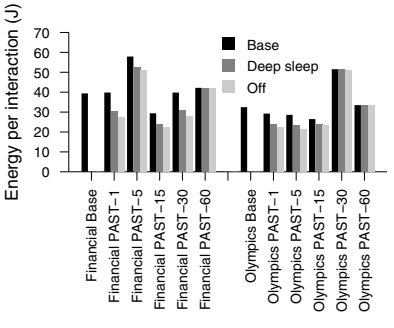


Fig. 9. PAST energy per Web interaction

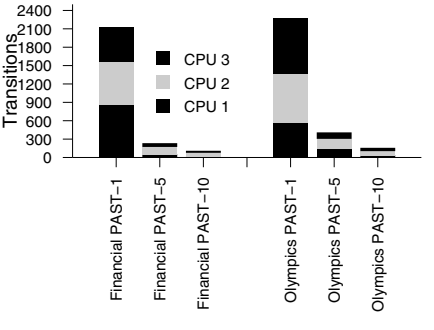


Fig. 10. PAST CPU transitions

Number of Transitions. Turning server-class CPUs on and off may create physical stress due to thermal expansion and contraction, potentially decreasing mean time to failure. We consider how much stress PAST may place on a system by examining the number of transitions between different system configurations. Each transition means that a CPU has been turned off or on and cycles have been wasted during the transition. We would like to minimize the transitions while still providing good performance and energy savings. We have previously selected a sampling interval of 1 second, but this places a potentially large strain on the system with transitions occurring approximately every 3 seconds. We believe this rate may be too high.

6.3 AVG< N >

We next explored the AVG< N > policy for various values of N. We studied the effects of using values of N of 2, 3, 4 and 5. These do not correspond to a sampling interval, but instead to how quickly the input from a previous sample is lessened relative to more recent contributions. Smaller values of N mean that older contributions decay more rapidly than with larger values of N. The only value of N which met the BestSellers WIRT criteria was $N = 2$. This result was true for both financial and Olympics profiles and is not discussed any further here.

We begin our AVG< N > analysis by examining the amount of time spent in the various configurations in comparison to the oracle. Figure 11 shows that financial matches the oracle well with $N = 2$ and Olympics matches most closely with $N = 3$. However, because of the BestSellers WIRT, our solution space has already been limited to $N = 2$ only.

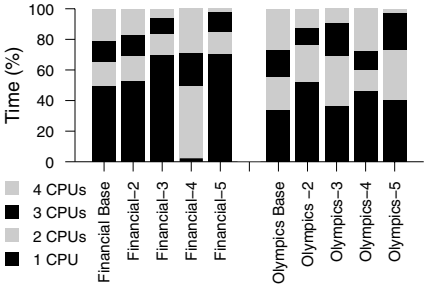


Fig. 11. AVG< N > configurations

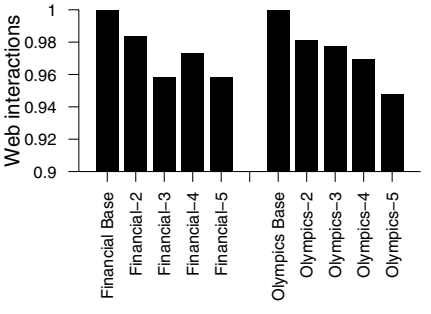


Fig. 12. AVG< N > throughput (normalized to base)

Figure 12 shows the relative number of interactions processed during the 60 minute experimental run. $N = 2$ shows a througput reduction of only 2%. This grows to a throughput loss of 7% with $N = 5$. The performance results indicate that AVG< N > may prove to be a good policy for CPU Packing.

Unfortunately, AVG< N > does not produce as large a savings in the energy per interaction as PAST does (Figure 13). The energy savings are still substantial so the policy is still viable and its applicability will probably be workload dependent. AVG< N > performs well for slower changes in utilization.

Finally, we examine the stress AVG< N > may place on the system in Figure 14. AVG< N >, even with $N = 2$, places far less stress on the system than PAST with a sampling interval of 1 second. AVG< N > forces only 1/3 as many transitions as PAST for policy parameters that meet the BestSellers WIRT criteria.

6.4 Hybrid

Both PAST and AVG< N > are promising policies, but we would like to combine the best aspects of each. In light of this, we chose to study a hybrid policy

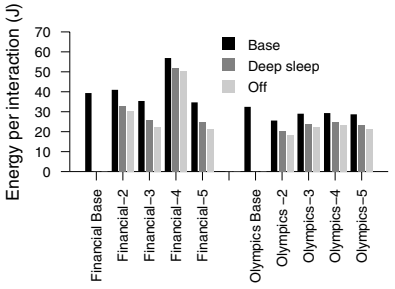


Fig. 13. AVG< N > Joules per Web interaction

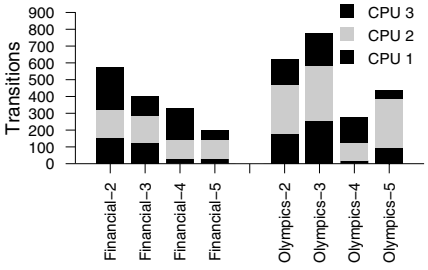


Fig. 14. AVG< N > transitions

that uses PAST to increase the number of CPUs and uses AVG< N > to decrease the number of CPUs. We found the hybrid reduced the stress on the system while only marginally decreasing the power savings. We selected PAST with a sampling interval of 1 and AVG< N > with $N = 2$ for our hybrid. It has performance penalties and power savings similar to PAST (3% performance loss, 25% reduction in joules per interaction), but only half as many transitions between system configurations.

7 Conclusions

Our results indicate that CPU Packing is a promising technique. It allows a server to respond to peak demands without paying the cost of constantly keeping all the CPUs on line. During periods of lower demand, running jobs are packed into fewer CPUs and the excess CPUs are placed into lower power states allowing the server to consume less power.

We found that PAST and AVG< N > were reasonable policies for CPU Packing. For each policy, we were able to identify a design point that met our restrictive BestSellers WIRT criteria. Our results are also applicable to systems with less stringent requirements where an increased delay in the response time may be acceptable in the short term as long as the system is able to recover quickly. Our hybrid policy combined the best elements of each policy to provide a solution which reduces system performance by only 3% while decreasing the energy per interaction by 25%. It also minimizes the number of transitions between the on and off/deep sleep states, reducing the potential strain on the system.

We plan to expand our work to include different architectural and microarchitectural configurations. In addition, we plan to study additional policies with more focus on predictive policies and the use of long duration phase history to enhance these policies. Finally, we plan to study the suitability of using CPU packing techniques on servers intended for interactive use rather than on those meant only to serve remote requests.

References

1. Felter, W., Keller, T., Kistler, M., Lefurgy, C., Rajamani, K., Rawson, F.L., Hensbergen, E.V.: Energy management for commercial servers. IEEE Computer (2003)
2. Weiser, M., Welch, B., Demers, A., Shenker, S.: Scheduling for reduced CPU energy. In: Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI). (1994) 13–23
3. Govil, K., Chan, E., Wassermann, H.: Comparing algorithms for dynamic speed-setting of a low-power CPU. In: Proceedings of the 1st Conference on Mobile Computing and Networking MOBICOM'95. (1995)
4. Pering, T., Burd, T., Brodersen, R.: The Simulation of Dynamic Voltage Scaling Algorithms. In: IEEE Symposium on Low Power Electronics. (1998)
5. Pering, T., Burd, T., Brodersen, R.: Voltage scheduling in the lpARM microprocessor system. In: Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'00. (2000)
6. Grunwald, D., Levis, P., Farkas, K.I., Morrey III, C.B., Neufeld, M.: Policies for dynamic clock scheduling. In: Operating Systems Design and Implementations. (2000)
7. Flautner, K., Reinhardt, S., Mudge, T.: Automatic performance-setting for dynamic voltage scaling. In: Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01. (2001)
8. Bohrer, P., Elnozahy, E., Keller, T., Kistler, M., Lefurgy, C., Rajamony, R.: The case for power management in web servers. In: Power-Aware Computing (Robert Graybill and Rami Melhem, editors). Kluwer/Plenum series in Computer Science. (2002)
9. Elnozahy, E.M., Kistler, M., Rajamony, R.: Energy-efficient server clusters. In: Proceedings of the Second Workshop on Power Aware Computing Systems (held in conjunction with HPCA-2002). (2002)
10. Rajamani, K., Lefurgy, C.: Request-distribution schemes for saving energy in server clusters. In: IEEE International Symposium on Performance Analysis of Systems and Software. (2003)
11. Council, T.P.P.: http://www.tpc.org/tpcw (2003)
12. Council, T.P.P.: TPC Benchmark W (Web Commerce) Specification Version 1.8. http://www.tpc.org/tpcw (2002)
13. Fleming, M., Mwaikambo, Z., Sarma, D., Russell, R.: Linux CPU Hotplug patch. http://www.kernel.org/pub/linux/kernel/people/rusty/ (2002)

Exploring the Potential of Architecture-Level Power Optimizations

John S. Seng[1] and Dean M. Tullsen[2]

[1] Cal Poly State University,
Dept. of Computer Science,
San Luis Obispo, CA 93407
jseng@calpoly.edu
[2] University of California, San Diego,
Dept. of Computer Science and Engineering,
La Jolla, CA 92093-0114
tullsen@cs.ucsd.edu

Abstract. This paper examines the limits of microprocessor energy reduction available via certain classes of architecture-level optimization. It focuses on three sources of waste that consume energy. The first is the execution of instructions that are unnecessary for correct program execution. The second source of wasted power is speculation waste – waste due to speculative execution of instructions that do not commit their results. The third source is architectural waste. This comes from suboptimal sizing of processor structures. This study shows that when these sources of waste are eliminated, processor energy has the potential to be reduced by 55% and 52% for the integer and floating point benchmarks respectively.

1 Introduction

Much research has been aimed at reducing the power consumption of processors, including high-performance general purpose microprocessors. Power consumption can be reduced at many levels, from the circuit level to the application level. Our research explores the limits of power reduction available due to optimizations at the architecture level. We identify three broad categories of wasted energy that either have been, or could potentially be, targeted for reduction. Those categories are program waste, speculation waste, and architectural waste. The focus of this research is the reduction of power and energy dissipation in high-performance, high instruction level parallelism processors. and that is the context in which these limits are studied.

Program waste arises when program execution contains instructions that are not necessary for correct execution. This includes instructions that produce either dead or redundant values, or any instruction whose only consumers produce dead or redundant values. We consider an instruction *unnecessary* if it produces a dead value or does not affect any change on processor state (e.g., the contents

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 132–147, 2004.

of registers and memory). Examples of power and performance optimizations that target this type of waste include elimination of silent stores [38, 41] and dynamically dead instructions [10].

Speculation waste results from speculative execution following mispredicted branches. These instructions are fetched into the processor and consume pipeline resources and energy, but are never committed and do not change permanent processor state. Examples of energy and performance optimizations that target this type of waste include pipeline gating [23] and aggressive branch predictors [32].

Architectural waste results from the static sizing of architectural structures, such as caches, instruction queues, branch predictors, etc. Memory array structures are often designed to hold a large amount of data in order to obtain good overall performance, but a given program usually cannot exploit this at all times. Instruction queues are designed with a large number of entries when often there is little parallelism in the running code. For performance reasons, these structures are made as large as possible. Making them too small can also waste power (causing more miss traffic, increasing misspeculation, etc.). This study examines the power dissipation of caches and instruction queues that are always exactly the right size. Examples of power and performance optimizations that target this type of waste include selective cache ways [1] and dynamically resized issue queues [14].

The analysis in this paper assumes an aggressive, dynamically scheduled wide superscalar processor. Because this is a limit study, we make no assumptions about how or to what extent these sources of waste are avoided. Some of the techniques to avoid the waste are obvious, or have been investigated before, others are more difficult to imagine. In this paper, we strive to not let our imagination constrain the evaluation of the limits of architectural power reduction.

This paper is organized as follows. Section 2 describes related work. Section 3 describes the simulation and experimental methodology used in our experiments. Section 4 describes program waste and its associated energy costs. Section 5 describes speculation waste and quantifies the cost of speculative execution. Section 6 describes architectural waste. Section 7 demonstrates the effect on energy consumption when the three sources of waste are removed. The paper concludes in Section 8.

This is the background for the power limit paper

2 Related Work

Program waste occurs when instructions that are either dead or redundant are executed by the processor. Prior works have studied particular types of unnecessary instructions. Specific prior work has targeted dynamically dead instructions [10, 24], silent stores [20, 38, 41], silent loads [21, 25, 38], silent register writes [16, 24, 37], and predictably redundant instructions [9, 33]. Each of these works provides a technique for dynamically identifying and predicting

unnecessary instructions. Many of them only study the performance gains from eliminating execution of the these instructions.

A further analysis of unnecessary instruction chains is provided in [30]. In [30], Rotenberg similarly identifies sequences of instructions that are executed, but do not contribute to program correctness. That work analyzes sequence lengths of unnecessary instructions and proposes predicting such sequences.

Speculation waste occurs when instructions are fetched and executed after a mispredicted branch. One approach to solving this problem is to improve the accuracy of branch predictors. This has been extensively studied. Work that attempts to minimize the energy effects of misspeculated instructions is much more limited. Pipeline gating [23] addresses the problem of misspeculation by keeping instructions from continuing down the processor pipeline once a low-confidence branch has been fetched. By limiting the number of instructions fetched after an often mispredicted branch, the amount of energy wasted can be minimized. A similar concept is presented by Bahar et al. [3]. Seng et al. [31] demonstrate that multithreaded execution [36] conserves a significant amount of energy by relying less on speculation to achieve performance.

Suboptimal sizing of processor structures exists when hardware structures are inappropriately sized, both statically and dynamically, for program behavior and the different phases of program behavior. An approach to solving this problem for caches has been dynamic reconfiguration of cache structures. This has been studied in [5, 19, 29, 39, 40]. Another general technique to reduce the power consumption of caches is to power down parts of the cache that are unused [1, 27].

Several papers have proposed techniques for reducing the energy consumption of the instruction issue logic. A common approach to reducing the energy of the issue queue has been to dynamically reconfigure the issue window size [11, 12, 14, 15]. The authors note that most instructions are issued from the head of the issue queue. They propose particular implementations of dynamically resized issue queues in order to gain power savings. Other work, in contrast, resizes the issue queue on the basis of available parallelism [4, 17].

Another technique to lessen the cost of suboptimal sizing is hierarchical designs. This model includes deeper cache hierarchies [2], as well as hierarchical instruction queues [28].

Some areas that we do not study include the branch predictor, the functional units, and the register file. Energy optimizations for branch predictors are studied in [26]. Some proposed architecture-level optimizations for functional units are described in [4, 7]. Register file optimizations include hierarchical register files [6].

3 Methodology

Simulations for this research were performed with the SMTSIM simulator [35], used exclusively in single-thread mode. In that mode it provides an accurate model of an out-of-order processor executing the Alpha instruction set architecture. The simulator was modified to include the Wattch 1.02 architecture level power model [8].

Table 1. The benchmarks (integer and floating point) used in this study, including inputs and fast-forward distances used to bypass initialization

Benchmark	Input	Fast forward (millions)
crafty	crafty.in	1000
eon	kajiya	100
gcc	200.i	10
gzip	input.program	50
parser	ref.in	300
perlbmk	perfect.pl	2000
twolf	ref	2500
vortex	lendian1.raw	2000
vpr	route	1000
art	c756hel.in	2000
equake	inp.in	3000
galgel	galgel.in	2600
gap	ref.in	1000
mesa	mesa.in	1000
mgrid	mgrid.in	2000

Table 2. The processor configuration modeled

Parameter	Value
Fetch bandwidth	8 instructions per cycle
Functional Units	3 FP, 6 Int (4 load/store)
Instruction Queues	64-entry FP, 64-entry Int
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache (on-chip)	1 MB, 4-way, 64-byte lines
Latency (to CPU)	L2 18 cycles, Memory 300 cycles
Pipeline depth	8 stages
Min branch penalty	6 cycles
Branch predictor	21264 predictor
Instruction Latency	Based on Alpha 21164

The SPEC2000 benchmarks were used to evaluate the designs, compiled using the Digital `cc` compiler with the `-O3` level of optimization. All simulations execute for 300 million committed instructions. The benchmarks are fast forwarded (emulated but not simulated) a sufficient distance to bypass initialization and startup code before measured simulation begins. Additionally, the processor caches and branch predictor are run through a warmup period of 10 million cycles before data collection. Table 1 shows the benchmarks used, their inputs, and the number of instructions fast forwarded. In all cases, the inputs were taken from among the reference inputs for those benchmarks.

Details of the simulated processor model are given in Table 2. The processor model simulated is that of an 8-fetch 8-stage out-of-order superscalar microprocessor with 6 integer functional units. The instruction and floating-point queues contain 64 entries each, except when specified otherwise. The simulations model a processor with instruction and data caches, along with an on-chip secondary cache.

For the experiments involving program waste, a trace of 300 million dynamic committed instructions is captured. Dynamic instruction instances are then marked in the trace if they are determined to be unnecessarily executed. Producers for these instructions are also considered unnecessary and marked if all their dependents are either unnecessary or producers for unnecessary instructions. In order to capture the dependence chain of instructions, the instruction trace is processed in reverse order.

In order to model the effect of removing the energy usage of dynamic instruction instances, the trace is used as input during a second simulation run. Care was taken to ensure that the same instructions were executed for each run of the simulator - this guaranteed that only the effects of the dead and redundant instructions were being measured. When an instruction is marked as unnecessary, we emulate it in the simulator but do not charge the processor for the energy cost of any aspect of its execution.

4 Program Waste

Many instructions that are executed do not contribute to the useful work performed by the program. These instructions may include ones which produce values which will never be used, those instructions which do not change processor state in any way, or which predictably change program state in the same way. In this work, we quantify the impact of these redundant instructions on the energy usage of a processor. We start by examining those instructions that do not do useful work (either produce dead values or do not change program state).

4.1 Unnecessary Instructions

Instructions which do not do useful work for program execution are considered *unnecessary*. We classify unnecessary instructions into a number of categories.

Dead instructions (*dead* in Figures 1 and 2). Instructions which produce dead values are instructions where the destination register will not be read before being overwritten by another instruction. In addition, store instructions are also considered dead if the memory location written is overwritten before being read by a load instruction. A dead instruction can produce a value which is dead for every execution instance (statically dead, because of poor instruction scheduling by a compiler) or for select instances (because of a particular path taken through the code).

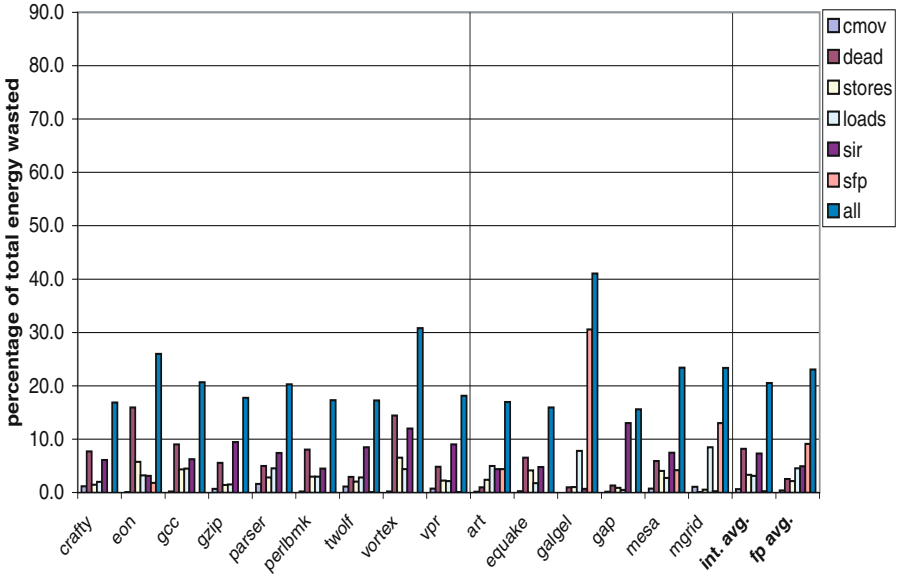


Fig. 1. Processor energy waste due to unnecessary instructions

Silent stores (*stores*). These are store operations where the value stored is the same as the value already at the address being stored to. Prior research has shown techniques which identify these instructions and remove them from the instruction stream [38, 41]. Removing silent stores from the dynamic instruction stream improves performance by reducing the memory traffic generated by these instructions.

Silent loads (*loads*). These are similar in nature to silent stores. Silent loads are load instructions where the value being loaded is the same as the value already in the destination register.

Silent register operations . Silent integer register operations (*sir*) are those integer instructions whose result is the same as the value already in the destination register. Similarly, silent floating point operations (*sfp*) are those instructions where the resulting value is the same as the destination floating point register.

Silent conditional moves (*cmov*). These are conditional moves where the condition is not met and the move does not occur.

In addition to the dynamic instruction instances determined to be unnecessary, in many cases the *producers* of values consumed by those instructions can be marked unnecessary as well, as long as the value only gets used by instructions already deemed to be unnecessary. Since we search the dynamic trace in reverse order, this can be determined with a single pass through the trace. In fol-

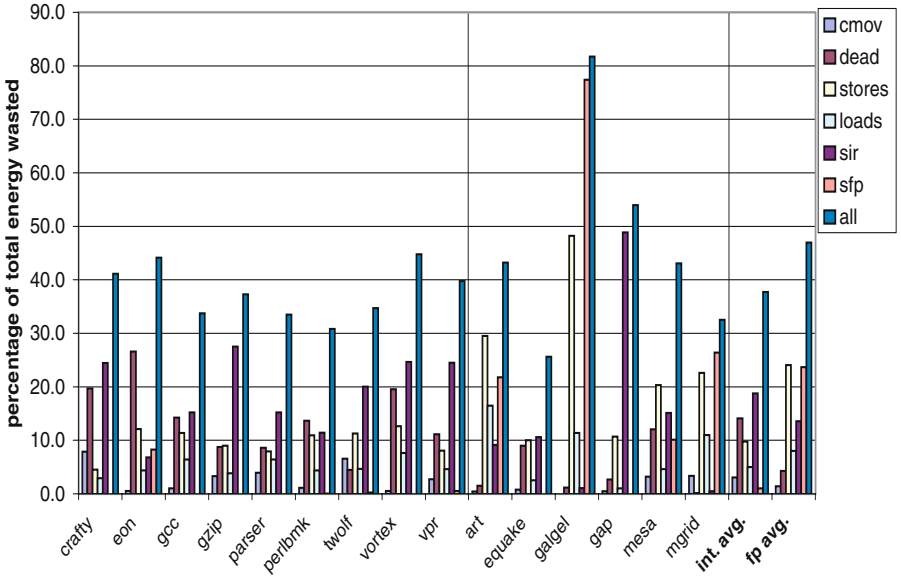


Fig. 2. Processor energy waste due to unnecessary instructions and their producers

lowing chains of unnecessary instructions, we only follow register dependences, assuming that values passed through memory are useful.

Program Waste Results. Figure 1 shows the percentage of total processor energy used for the execution of each type of program waste. The results shown represents the energy waste only due to those instructions which are marked as unnecessary and does not include the producers for those instructions. The data is shown for each of the integer benchmarks, the average of the integer benchmarks, each of the floating point benchmarks, and the average of the floating point benchmarks. The categories of unnecessary instructions are mutually exclusive except for the *dead* category where instructions are marked regardless of their type. The *all* category represents accounting for all types of program waste.

For the integer benchmarks, the most significant sources of program energy waste are due to dead (8.2%) and silent integer operations (7.4%). The `eon` benchmark wastes more energy with dead instructions (16.0%) than any other integer benchmark. Removing the energy costs of silent conditional operations provides little change (0.7% for the average). For the average of the integer benchmarks the total waste is 20.6%.

For the average of the floating point benchmarks, most waste is due to silent loads, silent integer, and silent floating point operations. Some of the floating point benchmarks exhibit very little waste for silent floating point operations (`art` and `gap`); for the `galgel` benchmark, this waste is significant at 30.6%. As

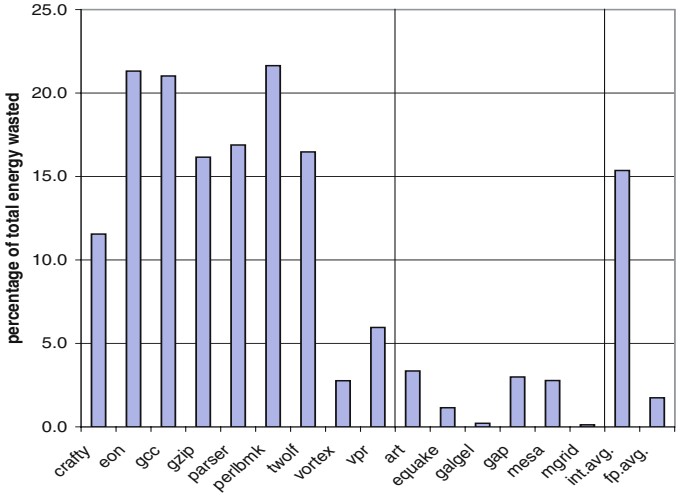


Fig. 3. Percentage of total energy that is wasted due to misspeculated instructions

with the integer benchmarks, little energy is wasted on silent conditional moves. The average total program waste for the floating point benchmarks is 23.1%.

When including the instructions whose only contribution is to produce values for unnecessary instructions (Figure 2), the potential energy savings increases significantly. The waste due to dead instruction chains is 14.1% and the waste due to silent integer instruction chains is 18.8%.

For the floating point benchmarks, silent floating point and silent store instruction sequences are the leading sources of energy waste. The `galgel` benchmark has an exceptionally large number of unnecessary silent floating point instruction chains (77.4% energy wasted). For these benchmarks, when counting the producer instructions for the silent stores, the total energy waste is significant (24.1%). This demonstrates that in the floating point benchmarks there are long sequences of instructions to compute a store value, and often that store is silent. In fact, for both integer and floating point programs, the category that gained (relatively) the most when including producing instructions was the silent stores.

It should be noted that the sum of the different sources of program waste are not strictly additive. Some may be counted in two categories (e.g., an instruction that is both redundant and dead). In fact, we see a high incidence of this effect when we include the producers, because many unnecessary instructions are also producers of other types of unnecessary instructions. We only count these instructions once for the *all* category. The total can also be more than additive, because a producer instruction may produce a value which is then consumed by multiple unnecessary instructions of different types, and is only considered unnecessary in the case that we are considering all its dependents as unnecessary, and thus only be counted in the *all* case.

These results indicate that the potential for energy savings is much more significant when including the chains of instructions that together produce an unnecessary value. Considering all unnecessary instructions, the integer benchmarks waste 37.7% of total energy. The total for the floating point benchmarks is greater at 47.0%.

5 Speculation Waste

For high performance, a long pipeline requires the prediction of branches and the execution of subsequent instructions. Figure 3 shows the effect on processor energy usage of misspeculation. In this experiment, we simulate the benchmarks 2 times. In the first simulation we obtain the energy results for all instructions entering the processor pipeline. In the second set of simulations, we only account for the energy of those instructions which are committed. The effect of the misspeculation is much more significant in the integer benchmarks because of the increased number of difficult to predict branches. For our benchmarks, there are on average 3.3 times as many mispredicted branches in the integer benchmarks as in the floating point benchmarks.

In this work, we have not studied the energy effects of other forms of speculative execution (e.g. load speculation [13] or value speculation [22]). We focus on control speculation as it will be an increasingly significant portion of total energy waste as processor pipelines increase in length.

6 Architectural Waste

In this section we study how the architectural design of a processor can contribute to the energy wasted during program execution. We define architectural waste to occur when the size of a processor structure is not optimal for the current state of the particular application currently executing. Suboptimal structure sizing occurs when a resource on the processor is larger (or in some cases smaller) than it needs to be for a particular program.

The structures we look at are the instruction and data caches, and the instruction issue queues. These represent a subset of all processor structures that could be adaptively sized, but do represent key structures which consume a significant amount of power and are typically sized as large as possible for peak performance. Other memory-based structures that could be studied in a similar manner include the register file, TLBs, reorder buffer, and the branch predictor. A study of optimal structure sizing provides insight into the potential energy savings to be gained given oracle knowledge of program behavior.

6.1 Cache Resizing

Because different programs have different memory behavior and access patterns, a cache of fixed size and configuration is not optimal for all cases. In cases where the processor is executing a tight loop, a large instruction cache is not

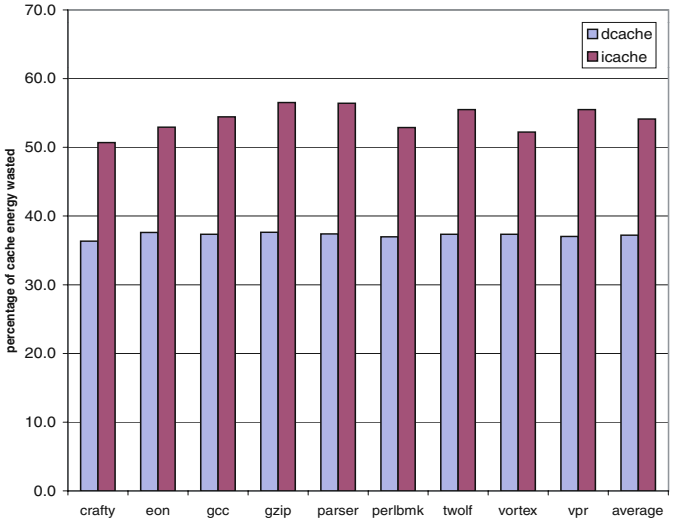


Fig. 4. Potential cache energy savings with optimal cache sizing for the integer benchmarks

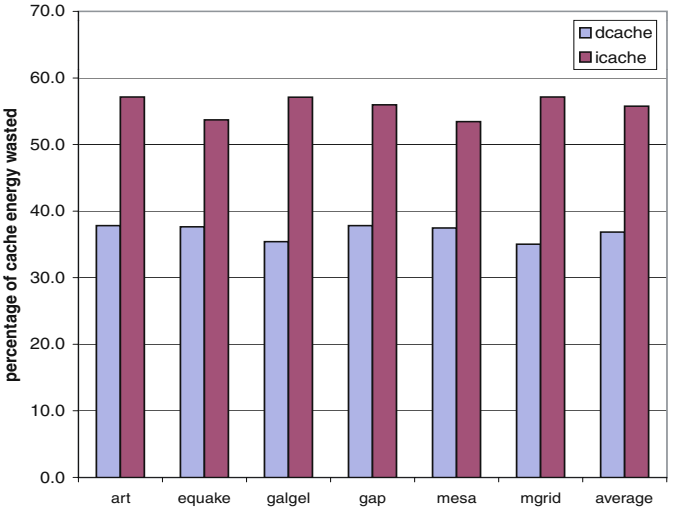


Fig. 5. Potential cache energy savings with optimal cache sizing for the floating point benchmarks

necessary. This is also true for the data cache if the memory footprint of the application is very small. In these instances smaller caches would provide the same performance, but with lower energy requirements.

In order to quantify the amount of energy wasted due to suboptimal cache sizes, we need to model a cache that is of the perfect size for the application currently being executed. To simulate the optimally sized cache, we simulate a continuum of cache sizes and configurations (24 different instruction and 24 different data caches). The cache sizes range from 2KB to 64KB with associativities ranging from 1 way to 8 way. Each cycle we select the smallest (lowest power) cache from among those (typically several) that hit on the most accesses. To compute the energy savings, we compare the optimal results against our baseline, a 64 KB 2-way cache.

In figure 4 we show the results of this experiment for the integer benchmarks. The results for the floating point benchmarks are shown in figure 5. The data represents the fraction of cache energy that is wasted due to a cache that is not of optimal size at the time of access. The energy modeled represents energy due to cache accesses and no other effects such as leakage.

The results show potential energy reductions for the integer applications of 54.1% of the total instruction cache power and 37.2% for the data cache. For the floating point applications, the potential reduction is 55.7% and 36.8% for the instruction and data caches, respectively.

6.2 Issue Queue Resizing

In an out-of-order processor, the size of the issue window greatly affects performance. A larger window size allows a processor to extract more parallelism from the code sequence it is currently executing. Although a large window is helpful, there are instances where the instruction stream contains much less parallelism

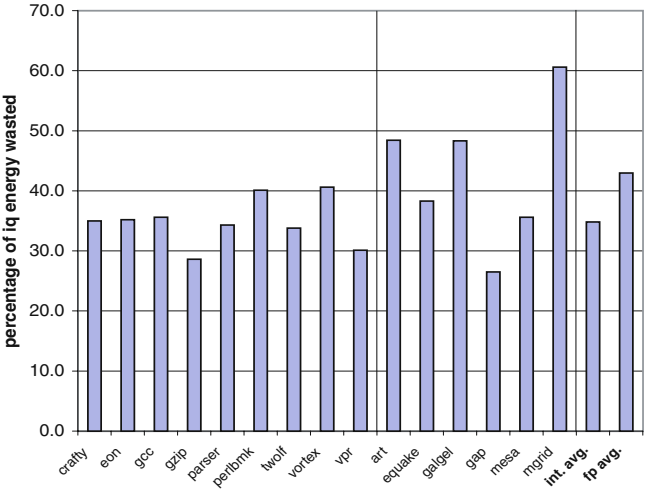


Fig. 6. Potential for energy savings available for optimally sized instruction queues

than the maximum processor issue width. Additionally, the issueable instructions are frequently located in a small portion of the issue queue [14]. In this case the wakeup logic and selection logic are much larger than necessary. It is these instances for which we consider an instruction queue design to be wasteful. This section quantifies that waste.

In order to measure the waste due to oversized issue queues, we compare the energy used by a fixed size issue queue versus one that can be dynamically resized. We assume that the dynamically resized queue is optimally resized every cycle. An optimally sized queue is one where the queue contains the fewest number of entries required to maintain the same performance (that is, issue the same number of instructions over that interval) as a large fixed size queue. In these experiments, the fixed size instruction queue contains 64 entries. The energy shown for the integer benchmarks is for only the integer queue because of the small number of floating point instructions in these benchmarks. The data shown for the floating point benchmarks is for the sum total energy of the integer and floating point queues.

Figure 6 shows the amount of instruction queue energy waste compared to an optimally sized queue. The fraction of instruction queue energy wasted for the integer benchmarks is 34.8% on the average. For the floating point benchmarks, the waste is 43.0%.

7 Removing Program, Speculation, Architecture Waste

We now show the results of removing program waste, speculation waste, and architecture waste. In these results, we run the simulations removing all energy costs associated with all the instruction types that are considered to be unnecessary computation. In addition, all energy due to speculation and suboptimal structure sizing are removed as well.

Figure 7 shows the results for the integer and floating point benchmarks, respectively. For the integer benchmarks, the overall energy waste ranges from 48.4% to 62.4%, with an average of 55.2%. For the floating point benchmarks, the overall energy savings ranges from 33.7% to 83.3%, with an average of 52.2%.

These results indicate that there are certainly significant gains to be had for architecture-level power optimizations. However, it also shows that for a very wide class of these optimizations, the total savings available are not huge. Order of magnitude decreases in energy are not going to come from exploiting wasted execution bandwidth and dynamic reconfiguration of conventional processors. They will need to come from more profound changes to the way we architect processors. Note that our analysis of dynamic resizing is not complete, as we ignore some structures that can be made adaptable. However, the effect of the other structures is not going to dramatically change the conclusion here.

Interestingly, the total energy wasted for the integer and floating point benchmarks is similar, but comes from different sources. For the integer benchmarks, speculation is much more of a factor than for the floating point benchmarks. In both type of benchmarks, the biggest contributor was due to program waste.

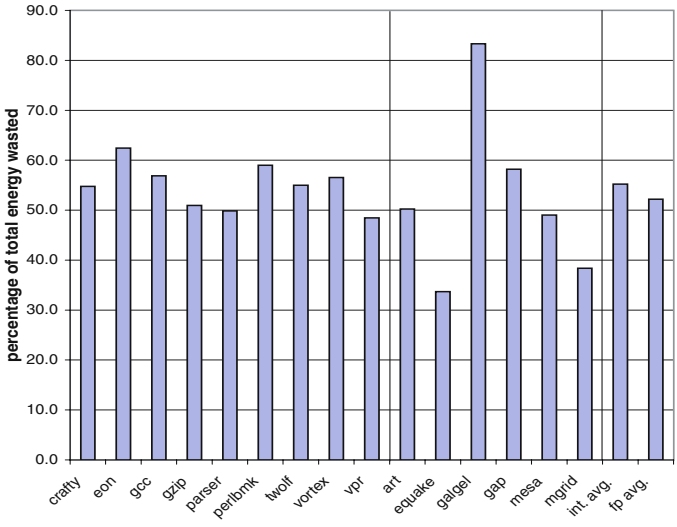


Fig. 7. Percentage of total energy wasted due to program and architectural waste

Regardless of how much speculation is removed from a processor or the sizing of processor structures, the energy cost of executing sequences of unnecessary instructions is large.

8 Conclusion

In this research, we examine the limits of architecture-level power reduction on a modern CPU architecture. We assume that there are three opportunities to save energy – do not do unnecessary computation (unnecessary speculation or dead computation), do not do redundant computation, and do not power architectural structures that are not needed (eliminate architectural waste via optimal sizing of structures).

This paper quantifies the energy lost to three sources of waste: program waste, speculation waste, and architectural waste. In the benchmarks we studied, we demonstrate that 37.7% of the energy used in executing the integer benchmarks is due to program waste. For the floating point benchmarks, 47.0% of the energy is because of program waste. It also shows that to take full advantage of program waste requires eliminating both the redundant instructions and their producers.

Speculation waste averages over 15% for the integer benchmarks, but is much lower for the floating point.

Architectural waste occurs when processor structures are larger than required and cannot be dynamically sized. An optimal instruction cache can consume 55% less energy than a 64KB instruction cache. Similarly, an optimal data cache can use 37% less than the baseline 64KB data cache.

When all of these sources of waste are eliminated, the total energy of the processor has the potential to be reduced by just over a factor of two for both the integer and floating point benchmarks. While this represents a significant opportunity, it also indicates that radical advances in power and energy efficiency require more significant architectural change than the adaptive techniques characterized by this limit study.

References

1. D. Albonesi. Selective cache ways: on-demand cache resource allocation. In *32nd International Symposium on Microarchitecture*, Dec. 1999.
2. R. I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 64–69, New York, Aug. 10–12 1998. ACM Press.
3. R. I. Bahar, G. Albera, and S. Manne. Using confidence to reduce energy consumption in high-performance microprocessors. In *International Symposium on Low Power Electronics and Design 1998*, Aug. 1998.
4. R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *28th Annual International Symposium on Computer Architecture*, May 2001.
5. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, Dec. 2000.
6. R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *34th International Symposium on Microarchitecture*, Dec. 2001.
7. D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA1999*, Jan. 1999.
8. D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, June 2000.
9. M. Burtsher and B. Zorn. Exploring last n value prediction. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1999.
10. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
11. A. Buyuktosunoglu, D. Albonesi, P. Bose, P. Cook, and S. Schuster. Tradeoffs in power-efficient issue queue design. In *International Symposium on Low Power Electronics and Design*, Aug. 2002.
12. A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems*, Nov. 2000.
13. B. Calder and G. Reinman. A comparative study of load speculation architectures. *Journal of Instruction Level Parallelism*, May, 2000.
14. D. Folegnani and A. Gonzalez. Reducing power consumption of the issue logic. In *Workshop on Complexity-Effective Design*, May 2000.
15. D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *28th Annual International Symposium on Computer Architecture*, June 2001.

16. M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *25th International Symposium on Microarchitecture*, Dec. 1992.
17. S. Ghiasi, J. Casmira, and D. Grunwald. Using ipc variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design*, June 2000.
18. S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *28th Annual International Symposium on Computer Architecture*, June 2001.
19. H. Kim, A. K. Somani, and A. Tyagi. A reconfigurable multi-function computing cache architecture. In *IEEE Transactions on Very Large Scale Integration Systems*, Aug. 2001.
20. K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *27th Annual International Symposium on Computer Architecture*, June 2000.
21. M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
22. M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
23. S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, June 1998.
24. M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *30th International Symposium on Microarchitecture*, Dec. 1997.
25. S. Onder and R. Gupta. Load and store reuse using register file contents. In *15th International Conference on Supercomputing*, June 2001.
26. H. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, Feb. 2002.
27. M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *34th International Symposium on Microarchitecture*, Dec. 2001.
28. S. Raasch, N. Binkert, and S. Reinhardt. A scalable instruction queue design using dependence chains. In *29th Annual International Symposium on Computer Architecture*, May 2002.
29. P. Ranganathan, S. Adve, and N. Jouppi. Reconfigurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, June 2000.
30. E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical report, North Carolina State University, 1999.
31. J. Seng, D. Tullsen, and G. Cai. Power-sensitive multithreaded architecture. In *International Conference on Computer Design 2000*, Sept. 2000.
32. A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, May 2002.
33. A. Sodani and G. Sohi. Dynamic instruction reuse. In *24th Annual International Symposium on Computer Architecture*, June 1997.
34. A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

35. D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.
36. D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip paralle lism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.
37. D. Tullsen and J. Seng. Storageless value prediction using prior register values. In *26th Annual International Symposium on Computer Architecture*, pages 270–279, May 1999.
38. J. Yang and R. Gupta. Energy-efficient load and store reuse. In *International Symposium on Low Power Electronic Design*, Aug. 2001.
39. S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Seventh International Symposium on High Performance Computer Architecture*, Jan. 2001.
40. S.-H. Yang, M. D. Powell, B. Falsafi, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Eighth International Symposium on High Performance Computer Architecture*, Feb. 2002.
41. A. Yoaz, R. Ronen, R. S. Chappell, and Y. Almog. Silence is golden? In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan. 2001.

Coupled Power and Thermal Simulation with Active Cooling

Weiping Liao and Lei He*

Electrical Engineering Department,
University of California, Los Angeles, CA 90095
{wliao, lhe}@ee.ucla.edu

Abstract. Power is rapidly becoming the primary design constraint for systems ranging from server computers to handhelds. In this paper we study microarchitecture-level coupled power and thermal simulation considering dynamic and leakage power models with temperature and voltage scaling. We develop an accurate temperature-dependent leakage power model and efficient temperature calculation, and show that leakage energy can be different by up to $10X$ for temperatures between 35^oC and 110^oC. Given the growing significance of leakage power and its sensitive dependence on temperature, no power simulation without considering dynamic temperature calculation is accurate. Furthermore, we discuss the thermal runaway induced by the interdependence between leakage power and temperature, and show that in the near future thermal runaway could be a severe problem. We also study the microarchitecture level coupled power and thermal management by novel active cooling techniques that reduce packaging thermal resistance. We show that the direct water-spray cooling technique reduces thermal resistance from $0.8^oC/W$ for conventional packaging to $0.05^oC/W$, and increases system maximum clock by up to $2.44X$ under the same thermal constraints.

1 Introduction

Power is rapidly becoming the primary design constraint for systems ranging from sever computers to handhelds, and the related thermal constraints are also emerging as an important issue. Thermal stress caused by high on-chip temperature and large temperature differentials between functional units may lead to malfunction of logic circuits, p-n junction breakdown, and clock skew [4] or ultimate physical failure of the microprocessor chip. Therefore, accurate power

* This paper is partially supported by NSF CAREER award CCR-0306682, SRC grant HJ-1008, a UC MICRO grant sponsored by Analog Devices, Fujitsu Laboratories of America, Intel and LSI Logic, and a Faculty Partner Award by IBM. We used computers donated by Intel and SUN Microsystems. Address comments to lhe@ee.ucla.edu.

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 148–163, 2004.

and thermal modeling is needed to develop and validate power and thermal optimization mechanisms.

As semiconductor technology scales to smaller feature sizes, leakage power increases exponentially because transistor threshold voltages are reduced in concert with supply voltage to maintain transistor performance. For current high-performance design methodologies, the contribution of leakage power increases at each technology generation [1], and the Intel Pentium IV processors running at 3GHz in 0.13um technology already have an almost equal amount of leakage and dynamic power [2]. The significance of leakage power exacerbates the thermal problems since leakage power has an exponential dependence on temperature [1]. Given this, power and thermal modeling is hardly accurate without considering the inter-dependency between leakage and temperature.

Almost all existing microarchitecture-level power simulators Wattch [6], SimplePower [5] and PowerImpact [7] do not consider temperature dependence of leakage power and assume a fixed ratio between dynamic and leakage power. [9] proposes a leakage power model with temperature dependence characterized by a purely empirical formula, and further applies the model for a cycle-accurate coupled power and thermal simulation. However, the temperature dependence is characterized by a purely empirical exponential term $exp(\frac{-a}{T-b})$ without providing a theoretical model, where a and b are coefficients and T is the temperature. Voltage scaling is not considered for either dynamic or leakage power in [9]. On the other hand, existing microarchitecture level thermal simulator HotSpot [17] models the thermal package such as spreader and heatsink and considers three dimensional heat transfer, but it fails to consider temperature dependency of leakage. In short, there is no existing simulator with accurate thermal modeling and accurate interdependence of temperature and leakage power.

In this paper, we present power models with clock, voltage, and temperature scaling based on the BSIM2 subthreshold leakage current model. We develop a coupled thermal and power microarchitecture simulator considering interdependence between leakage and temperature. With this simulator, we are able to accurately simulate the inter-dependence between power and temperature and evaluate microarchitectural power and thermal management techniques. We show the dramatic dependence of leakage power on temperature at the microarchitecture level based on the thermal resistance and chip area of Intel Itanium 2 processors within the temperature range between 35^oC and 110^oC. We also theoretically discuss thermal runaway induced by the interdependence between leakage power and temperature. These studies underscore the need for coupled power and thermal simulation. We further study the impact of active cooling techniques. We show that the direct water-spray cooling technique reduces thermal resistance from 0.8^oC/W for conventional packaging to 0.05^oC/W and increases system maximum clock by up to $2.44X$ under the same thermal constraints.

The rest of this paper is organized as follows. In Section 2, we develop dynamic and leakage power models with both voltage and temperature scaling. In

Section 3, we introduce both transient and stable-state temperature calculation and the experiments on thermal-sensitive energy simulations at microarchitecture-level. In Section 4, we present the the impact of coupled power and thermal management with novel active cooling techniques. We conclude and discuss future work in Section 5.

2 Power Model with Temperature and Voltage Scaling

We define three power states: (i) *active mode*, where a circuit performs an operation and dissipates both dynamic power (P_d) and leakage power (P_s). The sum of P_d and P_s is active power (P_a). (ii) *standby mode*, where a circuit is idle but ready to execute an operation, and dissipates only leakage power(P_s). (iii) *inactive mode*, where a circuit is deactivated by power gating [12] or other leakage reduction techniques, and dissipates a reduced leakage power defined as inactive power (P_i). A circuit in the inactive mode requires a non-negligible amount of time to wake up before it can perform an useful operation [7].

In cycle accurate simulations, power is defined as the energy per clock cycle. Therefore, P_d is equal to $\frac{1}{2} f_s C V^2$ where C is the switching capacitance, V is the supply voltage and f_s is the switching factor per clock cycle. In essence, P_d is the energy to finish a fixed number of operations during one cycle. Consistently, P_s is defined as $P_{so} * t$ where P_{so} is leakage power per second and t is the clock period. Same as P_s, $P_i = P_{io} * t$ is proportional to the clock period with P_{io} being reduced leakage power in the inactive mode.

2.1 Dynamic Energy with Voltage Scaling

Dynamic energy is consumed by charging and discharging capacitances. It is independent of temperature, but has a quadratic dependence with supply voltage. For VLSI circuits, the relationship between circuit delay and supply voltage V_{dd} is $delay \propto V_{dd}/(V_{dd} - V_T)^2$, where V_t is the threshold voltage. By assuming the maximum clock $f_{max} = 1/delay$, the appropriate supply voltage to achieve f_{max} can be decided by (1):

$$f_{max} \propto (V_{dd} - V_T)^2 / V_{dd} \tag{1}$$

Therefore, the dynamic energy for each cycle varies to achieve different f_{max}.

2.2 Leakage Estimation with Voltage and Temperature Scaling

Leakage Model with Temperature and Voltage Scaling. Similar to [9], we use the leakage power model for logic circuits as (2):

$$P_s = N_{gate} * I_{avg} * V_{dd} \tag{2}$$

where N_{gate} is the total number of gates in the circuit and I_{avg} is the average leakage current per gate. We further consider temperature and voltage scaling according to the BSIM2 model subthreshold current model [1] as shown in (3):

$$I_{sub} = Ae^{\frac{(V_{GS}-V_T-\gamma V_{SB}+\eta V_{DS})}{nV_{TH}}}\left(1-e^{-\frac{V_{DS}}{V_{TH}}}\right) \tag{3}$$

$$A = \mu_0 C_{ox}\frac{W}{L_{eff}}{V_{TH}}^2 e^{1.8} \tag{4}$$

where V_{GS}, V_{DS} and V_{SB} are the gate-source, drain-source and source-bulk voltages, respectively; V_T is the zero-bias threshold voltage, V_{TH} is the thermal voltage $\frac{kT}{q}$, γ is the linearized body-effect coefficient, η is the Drain Induced Barrier Lowering (DIBL) coefficient, μ_0 is the carrier mobility, C_{ox} is gate capacitance per area, W is the width and L_{eff} is the effective gate length.

From (3) we can see the temperature scaling for leakage current is $T^2 e^{-\frac{1}{T}}$, where T is the temperature, and the voltage scaling for leakage current is $e^{-(\alpha V_{dd}+\beta)}$, where α and β are parameters to be decided. Based on these observation, we propose the following formula for I_{avg} considering the temperature and voltage scaling:

$$I_{avg}(T, V_{dd}) = I_s(T_0, V_0) * T^2 * e^{\left(-\frac{\alpha * V_{dd}+\beta}{T}\right)} \tag{5}$$

where I_s is a constant value for the reference temperature T_0 and voltage V_0. The coefficients α and β are decided by circuit designs. Values for α and β as well as validation of (5) will be presented in Section 2.2.

We also improve the formula in [9] with better temperature and voltage scaling as shown in (6) - (8):

$$P_{so} = P_{circuits} + P_{cells} \tag{6}$$

$$P_{circuits}(T, V_{dd}) = (X * words + Y * word\_size) * V_{dd} * T^2 * e^{\left(-\frac{\alpha * V_{dd}+\beta}{T}\right)} \tag{7}$$

$$P_{cells}(T, V_{dd}) = (Z * words * word\_size) * V_{dd} * T^2 * e^{\left(-\frac{\gamma * V_{dd}+\delta}{T}\right)} \tag{8}$$

where P_{cells} is the leakage power dissipated by SRAM memory cells and $P_{circuits}$ is the power generated by the circuits such as wordline drivers, precharge transistors, and etc. $P_{circuits}$ essentially has the same format as (2) as $X * words + Y * word\_size$ in (7) can be viewed as N_{gate}, and the scaling in $P_{circuits}$ is the same as (5). P_{cells} is proportional to the number of SRAM memory cells. X, Y, Z, γ and δ in (7) and (8) are coefficients decided by circuit designs. Values for X, Y, Z, γ and δ as well as validation of (7) and (8) will be presented in Section 2.2.

Leakage Model Validation. We collect the power consumption for different types of circuits at a few temperature levels by SPICE simulations. We then obtain the coefficients in (5) - (8) by curve fitting. Table 1 summarizes the

Table 1. Coefficients in (5) - (8) for 100nm technology, where MTCMOS and VRC are the power gating techniques for logic and SRAM arrays, respectively

	Logic circuits				Memory based units		
	X	Y	α	β	Z	γ	δ
Power gating	3.5931e-12	1.2080e-11	-1986.1263	4396.0880	8.7286e-11	-443.2760	3886.2712
No power gating	5.2972e-10	1.7165e-9	-614.9807	3528.4329	5.2946e-10	-711.9226	3725.5342

Table 2. Comparison between our formula and SPICE simulation. I_{avg} and P_{so} are for logic circuits and SRAM arrays, respectively. The SRAM arrays are represented as "row number" x "column number". The units for I_{avg} and P_{so} are uA and uW, respectively

Circuit	Temperature (^{o}C)	V_{dd}	I_{avg} or P_{so}		abs. err. %
			formula	SPICE	
adder	100	1.3	0.0230	0.0238	3.74
	50	1.3	0.00554	0.00551	0.71
multiplier	100	1.3	0.0209	0.0217	3.83
	50	1.3	0.00493	0.00506	2.63
shifter	100	1.3	0.0245	0.0255	3.92
	50	1.3	0.00592	0.00585	1.32
SRAM 128x32	50	1.3	54.1	56.8	4.81
	50	1.0	21.62	22.31	3.07
SRAM 512x32	50	1.3	211.7	227.2	6.85
	50	1.0	84.41	88.83	4.98

coefficients for ITRS 100nm technology we used. Table 2 compares our high-level leakage power estimation for logic circuits and SRAM arrays with SPICE simulations in ITRS 100nm technology. We use different circuits and temperature during curve fitting and verification. The overall difference between our formulas and SPICE simulation is less than 7%.

3 Coupled Power and Thermal Simulation

3.1 Temperature Calculation

We develop the thermal model based on conventional heat transfer theory [13]. The stable temperature at infinite time can be calculated according to (9):

$$T = T_a + R_t * P \tag{9}$$

where T is the temperature, T_a is the ambient temperature, P is the power consumption, and R_t is the thermal resistance, which is inversely proportional to area and indicates the ability to remove heat to the ambient under the steady-

state condition. According to (9), the heat loss to ambient can be modeled as $P_o = (T - T_a)/R_t$.

The unbalance between total power consumption P and heat loss to ambient P_o leads to the transient temperature T characterized by (10):

$$P - P_o = C_t \dot{T} \tag{10}$$

where C_t is the thermal capacitance. By substituting P_o in (10) with $(T-T_a)/R_t$ we can get the differential equation (11):

$$R_t C_t \dot{T} + (T - T_a) = R_t P \tag{11}$$

where $\dot{T} = \Delta T/\Delta t$ and ΔT is the temperature change after a short time period Δt. By manipulating (11) we can get (12) for the temperature change ΔT:

$$\Delta T = \frac{PR_t - (T - T_a)}{\tau} \Delta t \tag{12}$$

where $\tau = R_t C_t$ is the thermal time constant. By solving (12) we can obtain an exponential form for temperature T in terms of time t and power, as shown in (13):

$$T = PR_t + T_a - (PR_t + T_a - T_0) \times e^{-\frac{t-t_0}{\tau}} \tag{13}$$

where T and T_0 are temperatures at two different time points t and t_0. This exponential form clearly shows that the power has a *delayed* impact on the temperature. Note that our cycle-accurate simulation uses (12) directly to avoid the time-consuming exponential calculation.

Same as [9], in our thermal model, we have two different modes with different granularities to calculate the temperature: (i) *individual mode*. We assume that there is no horizontal heat transfer between components, and calculate a temperature for each individual component. In general, the horizontal heat reduces the temperature gaps between components. So the individual mode essentially gives the upper bound of the highest on-chip temperature and temperature gap. (ii) *universal mode*, which is similar to the thermal model in TEM^2P^2EST [10]. We assume the whole processor as a single component with a uniform thermal characteristic and temperature. The universal mode gives the lower bound of the highest on-chip temperature.

3.2 Experiment Parameter Settings

Although our power and thermal models are applicable to any architecture, we study VLIW architecture in this paper. We integrate our thermal and power model into the PowerImpact [7] toolset. The microarchitecture components in

our VLIW processor include BTB, L1 instruction cache, L1 data cache, unified L2 cache, integer register file, floating-point register file, decoder units, integer units (IALUs) and floating-point units (FPUs). Among them, BTB, caches and register files are memory-based units, while the others are logic circuits. When calculating the power of memory-based units, we first partition the component into pieces of SRAM arrays with CACTI 3.0 toolset [14], then apply our formulas for power consumption of each SRAM array. The total component power consumption is the sum of power of all SRAM arrays. For IALUs and FPUs, we take the area and gate count in the design of DEC alpha 21264 processor [15], and scale from 350nm technology down to 100nm technology. For decode unit, we simply assume one decode unit has the same area and power consumption as one integer unit.

Table 3. System configuration for experiments

Component	Configuration			
Decode	6-issue width			
BTB	512 entries 4-way associative, Two-level predictor			
Register file	128 integer and 128 floating-point registers with 64-bit data width			
Memory	page size 4096 bytes, latency 30 cycles			
Memory Bus	8 bytes/cycle			
Functional units	Number	Latency		
Integer unit	4	1 cycle for add, 2 cycles for multiply and 15 cycles for division		
Floating-point unit	2	2 cycles for add/multiply, 15 cycles for division		
Cache	Size	Block size	Associativity	Policy
L1 Instruction	64 KB	32 bytes	4	LRU
L1 Data	64 KB	32 bytes	4	LRU
L2	2MB	64 bytes	8	LRU

To obtain a set of reasonable thermal resistances for components, we set the reference as the thermal resistance 0.8 $^oC/W$ for a chip with die size 374 mm^2 similar to Intel Itanium 2 [16]. Based on this reference, for each component, we calculate its thermal resistance as it is inversely proportional to its area. The whole chip thermal resistance is calculated in the same manner. Table 3 presents the micro-architecture configuration of the VLIW processors we study. Table 4 summarizes the power consumption, the thermal resistances and the areas for all components in our system. According to the thermal time constant for microarchitecture components without consider heatsink in [17], we set the thermal time constants as $\tau = 100us$, which is independent of component area.

To consider appropriate supply voltage scaling for varying clock, we assume that V_t is 20% of V_{dd} and $V_{dd} = 1V$ obtains 3GHz clock as specified by the ITRS. According to Equation (1) the corresponding V_{dd} for a range of clocks in our experiments is shown in Table 5.

Table 4. Power consumption (in pJ/cycle), thermal resistance R_t and areas for all components. For 100nm technology, we choose 1V supply voltage and 3GHz clock rate as specified by the ITRS. The decode, integer ALU and FPU are only one unit among total six, four and two units. The temperature is 35^oC. Note the P_s is relative small due to the low temperature

Component	P_a	P_s	P_i	R_t (K/W)	$Area$ (mm^2)
BTB	119	1.23	0.0504	64.4	1.63
L1 Instruction Cache	535	1.145	0.0458	22.129	4.74
L1 Data Cache	460	1.145	0.0458	20.967	4.99
Unified L2 Cache	1858	34.2	1.37	1.401	59.8
Integer Register File	59.6	0.027	0.0011	24.692	4.24
FP Register File	35.8	0.0275	0.0011	84.844	1.24
One Decode Unit	79.2	0.68	0.0068	236.355	0.44
One IALU	79.2	0.68	0.0068	236.355	0.44
One FPU	158	0.68	0.0068	125.599	0.83

Table 5. V_{dd} after appropriate voltage scaling for different clocks

Clock (GHz)	2	3	4	5
V_{dd} (V)	0.667	1.0	1.33	1.667

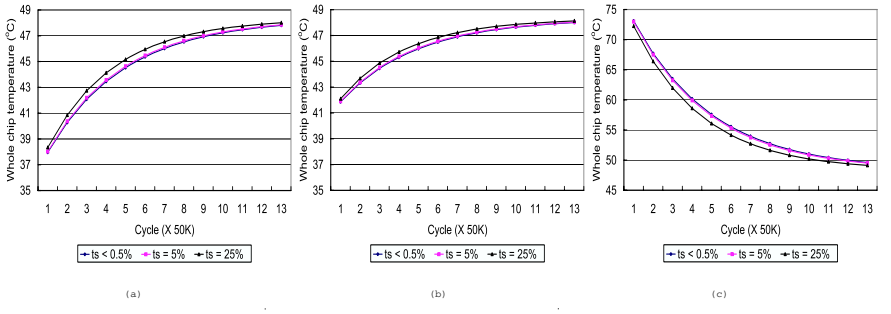


Fig. 1. Whole chip temperature curves obtained by the universal mode for different time step t_s. The clock frequency is 2GHz. Three different starting temperatures are chosen: (a) 35°C; (b) 40°C; and (c) 80°C. No throttling is applied. Therefore, the results are independent of benchmarks

3.3 Chip Temperature

In our experiments, we update temperatures after each time step t_s. We then update the power value with respect to new temperature for each t_s. Smaller t_s gives a more accurate transient temperature analysis, e.g., $t_s = 1$ cycle represents the cycle accurate temperature calculation. Figure 1 plots the transient

temperature for the whole chip calculated under different t_s shown as the percentages of the thermal time constant, where 0.5% of the thermal time constant is equal to 1000 clock cycles for a 2GHz clock. When $t_s \leq 0.5\%$ of the thermal time constant, the temperatures are identical to those with $t_s = 1$ cycle. Observable difference appears when t_s is increased to 5% of the thermal constants and significant error is induced when $t_s = 25\%$ of the thermal constant. Clearly, it is not necessary to update temperatures for each cycle. Since 0.5% of thermal constants always lead to negligible error on temperature calculation compared with the cycle accurate temperature calculation, we only update temperatures and power values after every period of 0.5% of the thermal time constants in the rest of the paper.

Note in Figure 1, we also present transient temperature with different starting temperatures. Clearly, different starting temperatures lead to the virtually same stable temperature without considering the thermal runaway problem which will be discussed in Section 3.5.

3.4 Temperature Dependent Leakage Power and Maximum Clock

Figure 2 shows the experimental results for total leakage energy consumption at 2.5GHz clock. We assume there is no throttling, i.e., P_a is dissipated in every cycle. We study two cases: one assumes a fixed temperature, and another considers energy consumption with temperature dependence in both individual mode and universal mode. From Figure 2 we can see that by changing the temperature from 35^{o}C to 110^{o}C, the total leakage energy can be changed by a factor of $10X$. Figure 2 clearly shows that any study regarding leakage energy is not accurate if the thermal issue is not considered. To consider temperature in methods in [7,18], the designers need to assume a fixed temperature appropriate for the processor and the environment, and then use leakage values at this temperature. How to decide the appropriate temperature is of paramount importance for accurate energy estimation, and it is an open problem in the literature. Our work actually presents an approach to select the appropriate temperature.

Faster system clock is always desired in the high-performance processor designs. However, as clock increases, the total energy and system temperature both increase as well. The maximum temperature and maximum temperature gap constraints prevent us from increasing the clock rate indefinitely. In the following experiments, we assume the maximum allowable temperature is 110^{o}C which is the maximum temperature supported by current semiconductor packaging techniques, and the maximum temperature gap among components is 40^{o}C. We use the individual mode to calculate the maximum temperature and the maximum temperature gap, where the maximum temperature is set as the largest temperature among all components [1]. Table 6 shows the maximum system temperature and the maximum temperature gap without any throttling. We can see that the maximum clock with thermal constraints is about 1.5GHz when there is no throttling.

[1] The universal mode gives us a lower bound of the maximum temperature.

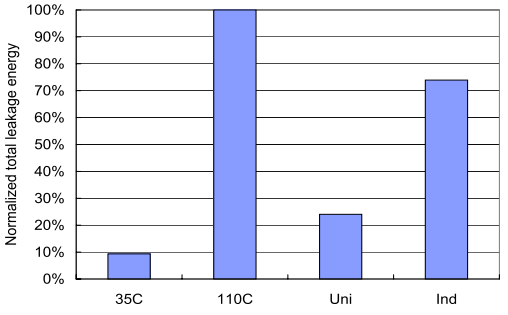


Fig. 2. Total Leakage energy consumption without any throttling. We study fixed temperatures of 35^{o}C and 110^{o}C, as well as the case with dynamically updated temperature. The cases of "ind" and "uni" stand for the individual mode and universal mode, respectively. The clock is 2.5GHz. Note the results are independent of benchmarks in the no-throttling cases

Table 6. Maximum temperatures (*Max T*) and temperature gaps (*Max Gap*) among components for different clocks without any throttling. The unit for temperatures is ^{o}C. The ambient temperature is 35^{o}C. Note the results are independent of benchmarks in the no-throttling cases.

Clock (GHz)	0.5	1	1.5	2	2.5
Max Temperature	35.2 - 36.016	36.7 - 41.5	40.7 - 56.7	48.4 - 87.3	61.4 - 157.2
Max Temperature Gap	0.92	3.97	19.19	46.23	110.44

3.5 Thermal Runaway

The MOSFET thermal runaway problem due to the positive feedback loop between the on-resistance, temperature and power of MOSFET is widely known [19]. In this section we will present another thermal runaway problem due to the interaction between leakage power and temperature. As the component temperature increases, its leakage power increases exponentially. The increase of power consumption further increases the temperature until the component is in thermal equilibrium with the package's heat removal ability. If the heat removal is not adequate, thermal runaway occurs as the temperature and leakage power interact in a positive feedback loop and both increase to infinity. For transient temperature T_0 and T_1 at consecutive time t_0 and t_1 and corresponding power $P(T_0)$ and $P(T_1)$, we define the following two criteria as necessary conditions for the thermal runaway to occur:

1. $T_1 > T_0$, i.e., the temperature should be increasing.
2. the increment of power is larger than the increment of package's heat removal ability. The package's heat removal ability is defined as $P_o(T) = \frac{T-T_a}{R_t}$ where T_a and R_t are ambient temperature and thermal resistance, respectively.

The second criterion can be mathematically formulated as (14) with relationship between T_0 and T_1 defined by (15):

$$P(T_1) - P(T_0) > \frac{T_1 - T_0}{R_t} \tag{14}$$

$$T_1 - T_0 = \frac{P(T_0)R_t - (T_0 - T_a)}{\tau}(t_1 - t_0) \tag{15}$$

where (15) is derived from (12).

In addition to temperatures, (14) and (15) require knowledge of runtime power, R_t, τ and T_a. We can simplify the second criterion with Theorem 1.

Theorem 1. *Criterion (2) is equivalent to* $\frac{d^2T}{dt^2} > 0$*, where* T *is temperature and* t *is time.*

The detailed proof of Theorem 1 can be found in [20]. □

Compared to (14) and (15), Theorem 1 provides a simpler mechanism with reduced complexity to detect thermal runaway.

We define the lowest temperature to meet the criteria 1 and 2 as *runaway temperature*. As long as the transient temperature reaches the runaway temperature, thermal runaway happens and the transient temperature will increase to infinity if no appropriate thermal management is applied. Figure 3 plots transient temperature curves with thermal runaway. [2] It clearly shows that as long as the transient temperature reaches the runaway temperature, thermal runaway occurs. Note two starting temperatures, 35^oC and 55^oC, are chosen in Figure 3. It is easy to see the starting temperature is independent of transient temperature behavior and thermal runaway is independent of the starting temperature because runaway temperature is decided by the power and the package's heat removal ability.

We calculate the runaway temperature according to criteria 1 and 2 for different clocks. Figure 4 shows the runaway temperatures for clocks from 4.5GHz to 6.5GHz. As clock increases, the runaway temperature decreases since the difference between power $P(T_1)$ and $P(T_0)$ increases. For clocks faster than 5.5GHz, the runaway temperatures of integer units are below our maximum temperature constraint 110^oC. In other words, we can not eliminate the thermal runaway by simply limiting the operating temperature to be no more than maximum junction temperature supported by current packaging techniques. We anticipate that thermal runaway could be a severe problem in the near future as the clock keeps increasing. Special thermal management schemes are expected to encounter this problem.

4 Power and Thermal Management with Active Cooling

As we can see from previous discussion, the designer's desire to increase system clock can be severely limited by thermal constraints. Better packaging and

[2] Memory units such as caches present similar curves and therefore are not shown.

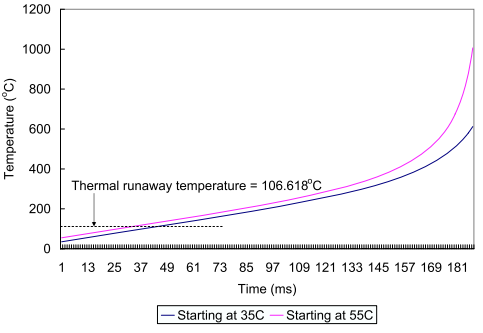


Fig. 3. Transient temperature curves one IALU with 5.5GHz clock. By reaching the runaway temperature, the thermal runaway happens and the transient temperature finally increases to infinity. The thermal runaway temperature is labeled. No throttling is applied

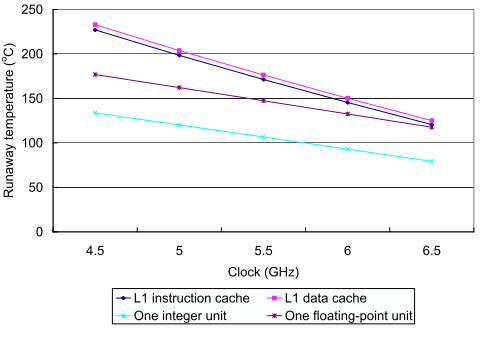


Fig. 4. Runaway temperatures for different clocks and different components

active cooling techniques can help to remove the thermal resistance, dissipate heat more quickly, and enable faster clocks. [4] discusses a few active cooling techniques such as cooling studs, microbellows cooling and microchannel cooling. [21] introduces a novel active cooling technique by direct water-spray cooling on electronic devices. In this section, we assume individual mode and consider three thermal resistance value: (i) $R_t = 0.8^o$C/W for the conventional cooling, (ii) $R_t = 0.05^o$C/W for water-spray cooling in [21], and (iii) $R_t = 0.45^o$C/W, a value in between the above two. We call both (ii) and (iii) as *active cooling* and study the impact of active cooling.

In our coupled power and thermal management, we turn off the clock signal for idle components by clock gating [22] and assume clock gating reduces 75% dynamic power. Our experiments show that clock gating can achieve the maximum clock 2.25GHz with the thermal resistance 0.8^oC/W under the same thermal constraints as those in Section 3. Furthermore, similar to [9], we evenly

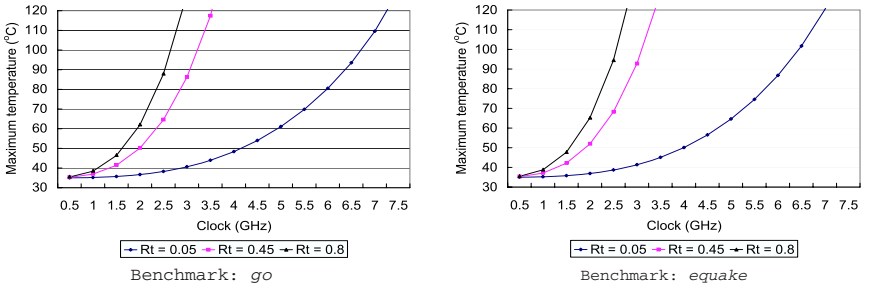


Fig. 5. Maximum temperature under individual mode with different thermal resistance

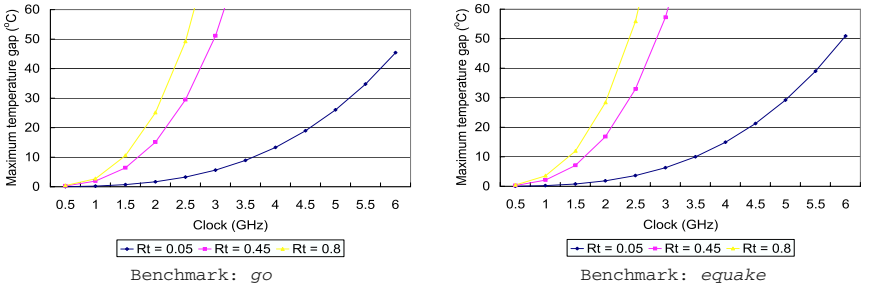


Fig. 6. Maximum temperature gap under individual mode with different thermal resistance

distribute instructions to functional units and eliminate the temperature gaps between integer units.

4.1 Maximum Clock

Figure 5 plots maximum temperatures for different clocks with different R_t. Obviously by applying active cooling techniques we can effectively increase the maximum clock while limiting the system temperature well below the thermal constraints. Figure 6 plots the maximum temperature gaps under different cooling techniques and clocks. By combining results in Figure 5 and 6 with the thermal constraints applied in Section 3.4, we can increase system clock to up to 5.5GHz by scaling V_{dd} up with $R_t = 0.05^o$C/W. Compared to the 2.25GHz maximum clock with $R_t = 0.8^o$C/W, the active cooling technique with $R_t = 0.05^o$C/W can increase the maximum clock by the factor of $2.44X$ under the same thermal constraints.

4.2 Total Energy

Figure 7 shows the total energy consumption with three different thermal resistances R_t. Clearly the cooling techniques substantially reduce the total en-

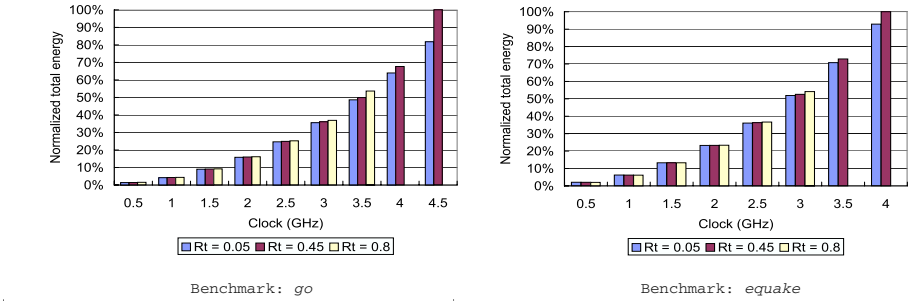


Fig. 7. Total energy consumption under individual mode with different thermal resistances. Note a few bars for clock at 3.5GHz and 4GHz are missing due to thermal runaway

ergy at the same clock. Compared to R_t of 0.45, R_t of 0.05 reduces the total energy by up to 18%. From Figure 7 we can also see that the energy reduction with active cooling techniques increases as clock increases, which means active cooling techniques is more effective for faster clocks. Note that in Figure 7 a few bars for $R_t = 0.45$ and $0.8^oC/W$ are missing due to thermal runaway. Traditionally the active cooling techniques such as cooling stubs and microchannel cooling [4] are only applied to mainframes computers. Our result clearly indicates that they can also be effective and may become necessary for microprocessors.

5 Conclusions and Discussions

Considering cycle accurate simulation, we have presented dynamic and leakage power models with clock, supply voltage and temperature scaling, and developed the coupled thermal and power simulation at the microarchitecture level. With this simulator, we have shown that the leakage energy can be different by up to $10X$ for different temperatures. Hence, microarchitecture level power simulation is hardly accurate without considering temperature dependent leakage model. We have studied the thermal runaway problem induced by interdependence between leakage power and temperature, and show that it could be a severe problem in the near feature as the runaway temperature can be much lower than the maximum temperature packages can support. We have studied the microarchitecture level coupled power and thermal management by novel active cooling techniques. We show that under the same thermal constraints, active cooling techniques such as water-spray cooling that reduces thermal resistance from $0.8^oC/W$ for conventional packaging to $0.05^oC/W$ can increase the maximum clock by a factor of $2.44X$. In this paper, we use lumped thermal model without distinguishing packaging components such as heat spreaders and heatsinks. In fact, we have developed a coupled power and thermal simulator PTscalar, which integrates temperature and voltage scalable leakage model

with accurate thermal calculation considering three dimensional heat transfer and the packaging components such as heat spreaders and heatsinks. This tool is available at http://eda.ee.ucla.edu/PTscalar. We believe that conclusions in this paper are still valid under the new PTscalar tool.

Acknowledgments

The authors would like to thank Dr. Kevin Lepak at Advanced Micro Devices, Inc. for helpful discussions.

References

1. A. Chandrakasan, W. J. Bowhill, and F. Fox, *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.
2. A. S. Grove, "Changing vectors of moore's law," in *Keynote speech, International Electron Devices Meeting*, Dec 2002.
3. T. Burd, T. Pering, A. Stratakos, and R. Bordersen, "A dynamic voltage-scaled microprocessor system," in *2000 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb 2000.
4. H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, 1990.
5. W.Ye, N.Vijaykrishnan, M.Kandemir, and M.J.Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *DAC*, 2000.
6. D.Brooks, V.Tiwari, and M.Martonosi, "Wattch: A framework for architectural-level power analysis optimization," in *ISCA*, 2000.
7. W. Liao, J. M. Basile, and L. He, "Leakage power modeling and reduction with data retention," in *ICCAD 02*, Nov 2002.
8. J. Butts and G. Sohi, "A static power model for architects," in *Proc. of MICRO33*, December 2000.
9. W. Liao, F. Li, and L. He, "Microarchitecture level power and thermal simulation considering temperature dependent leakage model," in *ISLPED*, Aug 2003.
10. A. Dhodapkar, C. Lim, and G. Cai, "Tem^{2}p^{2}est: A thermal enabled multi-model power/performance estimator," in *Workshop on Power Aware Computer Systems*, Nov 2000.
11. K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.
12. S. Mutoh and *et al*, "1-v power supply high-speed digital circuit technology with multithreshold-voltage cmos," *IEEE Journal of Solid-state circuits*, vol. 30, pp. 847–854, Aug. 1995.
13. J. V. D. Vegte., *Feedback Control System, 3rd Edition*. Prentice Hall, 1994.
14. P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," in *WRL Research Report 2001/2*, 2001.
15. B. A. Gieseke and et al, "A 600mhz superscalar risc microprocessor with out-of-order execution," in *Proc. IEEE Int. Solid-State Circuits Conf.*, pp. 176–177, 1997.
16. J. Stinson and S. Rusu, "A 1.5ghz third generation itanium 2 processor," in *DAC*, June 2003.

17. K. Skadron, T. Abdelzaher, and M. Stan, "Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management," in *Proceedings of the Eighth International Symposium on High-Performance Computer Architercture*, 2002.
18. H. Hanson, M. Hrishikesh, V. Agarwal, S. Keckler, and D. Burger, "Static energy reduction techniques for microprocessor caches," in *Proceedings of the International Conference on Computer Design*, 2001.
19. R. Severns, "Safe operating area and thermal design for mospower transistors," in *Siliconix applications note AN83-10*, Nov 1983.
20. W. Liao and L. He, *Microarchitecture Level Power and Thermal Simulation Considering Temperature Dependent Leakage Model.* Technical Report 04-246, University of California at Los Angeles, 2003.
21. M. Shaw, J. Waldrop, S. Chandrasekaran, B. Kagalwala, X. Jing, E. Brown, V. Dhir, and M. Fabbeo, "Enhanced thermal management by direct water spray of high-voltage, high power devices in a three-phase, 18-hp ac motor drive demonstration," in *Eighth Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, 2002.
22. V. Tiwari, D. Singh, S. Rajgopal, and G. Mehta, "Reducing power in high-performance microprocessors," in *DAC*, 1998.

The Synergy Between Power-Aware Memory Systems and Processor Voltage Scaling

Xiaobo Fan, Carla S. Ellis, and Alvin R. Lebeck

Department of Computer Science, Duke University, Durham, NC 27708, USA
{xiaobo, carla, alvy}@cs.duke.edu

Abstract. Energy consumption is becoming a limiting factor in the development of computer systems for a range of application domains. Since processor performance comes with a high power cost, there is increased interest in scaling the CPU voltage and clock frequency. Dynamic Voltage Scaling (DVS) is the technique for exploiting hardware capabilities to select an appropriate clock rate and voltage to meet application requirements at the lowest energy cost. Unfortunately, the power and performance contributions of other system components, in particular memory, complicate some of the simple assumptions upon which most DVS algorithms are based.

We show that there is a positive synergistic effect between DVS and power-aware memories that can transition into lower power states. This combination can offer greater energy savings than either technique alone (89% vs. 39% and 54%). We argue that memory-based criteria—information that is available in commonly provided hardware counters—are important factors for effective speed-setting in DVS algorithms and we develop a technique to estimate overall energy consumption based on them.

Keywords: Power-Aware, Memory System, DVS, Control Policy, Synergy.

1 Introduction

Energy consumption is becoming a limiting factor in the development of computer systems for a range of application domains – from mobile and embedded devices that depend on batteries to large hosting centers that incur enormous electricity bills. In recognition that the exponential growth in the performance of processors may come at a high power cost, there has been considerable interest in scaling the CPU supply voltage and clock frequency. Thus, if application demand does not currently need the highest level of processor performance, a lower power design point can be chosen temporarily. The excitement surrounding voltage/frequency scaling is based on characteristics of the power/performance tradeoffs of CMOS circuits such that the power consumption changes linearly with frequency and quadratically with voltage, yielding potential energy savings for reduced speed/voltage.

Dynamic Voltage Scaling (DVS) is the technique for exploiting this tradeoff whereby an appropriate clock rate and voltage is determined in response to dynamic application behavior. This involves two issues: predicting future processing needs of the workload and setting a speed (and associated voltage) that should satisfy those performance needs

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 164–179, 2004.

at the lowest energy cost. A number of DVS algorithms have been proposed [28, 24, 23, 13, 9, 7, 26, 8], primarily addressing the prediction issue. Most simulation-based studies of these algorithms have focussed solely on CPU energy consumption and have ignored both the power and performance contributions of other system components.

The importance of considering other system components is supported by the few studies based on actual implementation of DVS algorithms for which overall energy measurement results have been disappointing compared to simulation results. This has been attributed to several factors, including inaccuracies in predicting the future computational requirements of real workloads for those solutions based primarily on observations of CPU load and the inclusion of other components of the system beyond the CPU, especially interactions with memory [9, 7, 6, 19, 20, 21, 26]. Thus, the impact of memory has been considered to be a complicating factor for the straightforward application of DVS.

In this paper, we identify a positive synergy between voltage/frequency scaling of the processor and newer memory systems that offer their own power management features. Based on our simulation results, this paper makes the following contributions:

- We demonstrate that effective power-aware memory policies enhance the overall impact of DVS by significantly lowering the power cost of memory relative to the CPU. We discuss what it means to have an "energy-balanced" system design. The implication of a balanced system using traditional full-power memory chips with modern low-power, DVS-capable processors is that memory energy may dominate processor energy such that the overall impact on system energy of employing DVS is marginal. By better aligning the energy consumption of the processor and memory, the individual power management innovations of each device can produce greater benefits.
- We explore how different power-aware memory control policies affect the frequency setting decision. The synergy between DVS and sophisticated power-aware memory goes deeper than achieving a lower power design point. Even the simplest memory power management strategy that powers down the DRAM when the processor becomes idle introduces a tradeoff between CPU and memory energy that may negate the energy saving benefits of reducing the CPU frequency/voltage beyond some point. Thus, the lowest speed setting of the processor may not deliver the minimal combined energy use of processor and memory.
- We develop a technique to estimate overall energy consumption using information available from existing performance counters and show that our estimator is sufficient to capture the general trend in overall energy as CPU frequency changes. Given the energy tradeoffs inherent with a power-aware memory, we argue that the memory access behavior of the workload must be understood in order for the DVS system to predict the energy and performance implications of a particular frequency/voltage setting.

The remainder of this paper is organized as follows. The next section discusses background and related work. Section 3 describes our research roadmap and methodology, and Section 4 examines the interactions between DVS and a traditional high power, low latency memory design, confirming previous observations in the context of our environ-

ment. We examine the effects of power-aware memory and develop a memory-based estimator of overall energy in Section 5. We conclude in Section 6.

2 Background and Related Work

This section summarizes previous work on DVS. We also provide background on power aware memory.

2.1 Dynamic Voltage Scheduling

Dynamic voltage scheduling has been studied for a wide variety of workloads, including interactive, soft real time, and hard real time applications. Each of these workloads may require a different type of DVS algorithm based on the information available about the tasks, the tolerance for missed deadlines, and the nature of the application behavior. In general, most DVS algorithms divide total execution time into task periods [25, 13, 27, 11, 17] or regular intervals [28, 24, 9, 8] and attempt to slow down computation to just fill the period without missing the deadline or carrying work over into the next interval. The algorithm must predict the processing demands of future periods, usually from observed past behavior, and use that information to determine the appropriate processor speed and corresponding voltage. Recent work that falls somewhere in between the hard real time and the interval-based categories acknowledges the need for more semantic information about the workload to increase prediction of task execution demands [7, 6, 18, 26, 22]. These studies provide the rationale for our assuming good predictions for a specific workload.

The speed-setting decision has appeared to be straightforward, given good predictions. However recent experimental work [19, 26, 9, 10] has suggested that memory effects should be taken into account. For computations that run to completion, Martin [19, 21] shows there is a lower bound on frequency such that any further slowing degrades a metric defined as the amount of computation that can be performed per battery discharge. In recognition of the interaction between CPU and memory, Hsu [10] proposes a compiling technique to identify program regions and set appropriate CPU frequencies for them. For periodic computations, Pouwelse [26] alludes to the problem that the high cost of memory, extended over the whole period, may dominate the overall energy consumption of a system such that even effective DVS of the CPU delivers marginal benefit. We confirm this observation in the context of our target environment and then focus on memory technology that ameliorates the problem.

2.2 Power-Aware Memory

Previous work on power-aware memory systems [14, 3, 4, 5] introduces another complication such that the power consumption of memory varies significantly depending on how effectively the system can utilize a set of power states offered by the hardware. Power-aware memory chips can transition into states which consume less power but introduce additional latency to transition back into the active state. The lower the power consumption associated with a particular state, the higher the latency to service a new

memory request. We adopt a three-state model consisting of *active, standby*, and *powerdown* states with power and latency values as shown in Table 1 (based on Infineon Mobile-RAM [12]). We use 90ns as the average delay of accessing a 32-byte cache block. Additional delay (denoted by the +) is incurred for clock resynchronization.

Table 1. Mobile-RAM Power State and Transition Values

Power State or Transition	Power (mW)	Time (ns)
Active	$P_a = 275$	$t_{access} \approx 90$
Standby	$P_s = 75$	-
Powerdown	$P_p = 1.75$	-
Stby $\rightarrow$ Act	-	$T_{s \rightarrow a} = 0$
Pdn $\rightarrow$ Act	$P_{p \rightarrow a} = 138$	$T_{p \rightarrow a} = +7.5$

The memory controller can exploit these states by implementing dynamic power state transition policies that respond to observable memory access patterns. Such policies often are based on the idle time between runs of accesses (which we refer to as *gaps*) and threshold values to trigger transitions. Fig. 1 shows how a policy that transitions among *active*, *standby*, and *powerdown* modes might work. When the memory has outstanding requests, the memory chip stays active. Upon the completion of servicing requests, the chip automatically goes to *standby*. Note there is no additional latency to transition back to *active* from *standby*. When the idle time, gap, exceeds a threshold (e.g., $gap_i > Th$), the chip transitions to *powerdown* and stays there until the start of the next access. For gaps shorter than the threshold (e.g., $gap_j < Th$), the memory remains in *standby*.

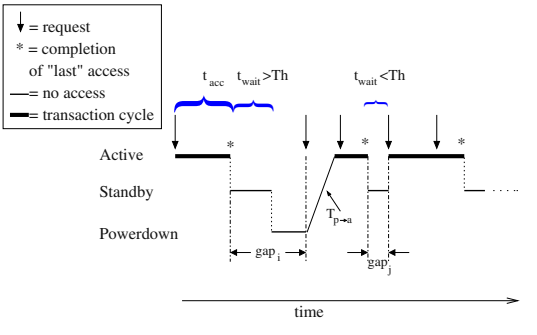


Fig. 1. Memory Access and Power Control

Operating system page allocation policies that place actively used pages in the minimum number of DRAM chips fully exploit the capabilities of power-aware memory. Previous studies [14] show that using a sequential first-touch virtual to physical page allocation policy enables unused DRAM chips to enter the powerdown state. Sequential page allocation, by concentrating all memory references to the minimum number of DRAM chips, produces significant energy savings over random page allocation.

3 The Synergy Between DVS and Power-Aware Memory

In this section, we first establish a roadmap for investigating the impact of DVS and power-aware memory on each other and how they can adapt to be mutually advantageous. Second we introduce our experimental environment and workload selection.

3.1 Roadmap

The primary goal of our study is to explore power-aware memory's influence on selecting the appropriate frequency/voltage to achieve the lowest energy use while satisfying a known need for a particular level of performance. Therefore, we focus on the factors that affect the speed-scaling decision in meeting those energy/performance goals. We also explore the influence of speed-scaling on the decisions of the power-aware memory controller. Since most DVS algorithms divide total execution time into task periods or episodes, through this paper all problems are discussed in the time domain of one task period.

We first consider a base case memory design in which the chips only transition between *active* and *standby*. It is the standard operating mode of most current DRAMs. We refer to this case as *standard memory*. A meager step in the direction of power-awareness is called *naive powerdown* and represents the policy in which the memory chips operate normally until task completion at which point they are powered down through the slack time to the end of the period. Next we explore memory controller policies that potentially transition a chip to a lower power state when it is otherwise not servicing request. This is done during the task's whole period (execution and slack time). The controller might wait for a threshold amount of time in *standby* before making the transition. According to previous research [14, 4, 5] and our experiments, the immediate transition and sequential page allocation represent "best practice". We refer to this policy as *immediate powerdown* or *aggressive*.

The other question we need to address is how the DVS algorithm can map the known performance needs of a task into a frequency range that can meet those needs when memory policies and behavior may have an effect on that performance. We explore the variation in execution times, defined as the busy portion of our experimental period, across the frequency range to understand the factors that the DVS algorithm must take into account.

3.2 Methodology

We use a modified version of the PowerAnalyzer [1] simulator from the University of Michigan for our experiments. We modified the simulator to include a detailed Mobile-RAM memory model including the power state transitions described in Section 2. We use two memory chips with a total capacity of 64MB. The variable voltage processor we simulate is based on Intel's XScale [16]. The voltage and frequency values used in our evaluations range from 50MHz and 0.65V to 1000MHz and 1.75V. The power consumption of the CPU at a given frequency/voltage setting is derived in the simulator from actual processor and cache activity. It varies significantly from approximately 15mW at 0.65V up to 2.2W at 1.75V. The 1-level on-die cache is 32KB with 32B blocks

Table 2. Variable Voltage Processor Values

Voltage (V)	**Frequency** (MHz)	**Power** (approximate mW)
0.65	50	15
0.70	100	33
0.80	200	80
1.00	400	222
1.20	600	434
1.40	800	732
1.75	1,000	1,300

and 32-way associativity. On average it takes about 90ns to service a cache miss without incurring the memory state transition delay.

As shown in Table 1, we use an SDRAM based memory model. In particular, we use close-page policy during the transition from $active$ to $standby$. Note data is not lost in the $powerdown$ state due to the refresh operation. We assume the impact of the refresh on the gap is minor since the refresh cycle is several orders of magnitude longer than the gap (ms vs. ns). For completeness, we also used a Rambus-DRAM based four-state model and obtained qualitatively similar results. Due to the similarity of results and the popularity of SDRAM based memory platforms (i.e., Mobile-RAM, DDR RAM, etc.), we only present results for the Mobile-RAM memory model.

We consider multimedia applications as representative workloads for low power mobile devices and because they appear to be amenable to good predictions of future processing demands on a per-task basis [26]. We have performed experiments using four applications from the MediaBench suite [15, 2]: MPEG2dec – an MPEG decoder, PEGWIT – a public key encryption program, G721 – voice compression, and RASTA – speech pre-processor. The results from these four benchmarks are remarkably consistent. Therefore we present results primarily for the MPEG decoder running at 15 frames per second (a period of 66ms). We use an input file of 3 frames consisting of one I-frame (intra-coded), one P-frame (predictive) and one B-frame (bidirectional). We present results for the P-frame, the other frames produce similar results. At this frame rate, decoding a single frame at our slowest frequency of 50MHz nearly fills the designated period for all of our experiments. Since the period of our application is set to match its execution time at 50MHz, we can explore energy consumption over the full range of available voltages without concern for missed deadlines. One way of viewing this is that the candidate frequencies which can deliver adequate performance have already been identified so the question of which voltage delivers the best results for our energy metrics can be fully explored.

To further explore those memory effects in a controlled fashion, we use a synthetic benchmark that can model a variety of computation times and cache miss ratios. For each miss ratio targeted, the synthetic benchmark is configured to just accommodate the execution of one task at 50MHz while barely meeting its deadline. We choose a 30ms period and target 3 miss ratios of 2%, 9% and 16%.

4 DVS and Standard Memory

We begin by taking standard Mobile-RAM as our base. The memory chip stays in *active* mode servicing requests and transitions to *standby* upon the completion of servicing requests. For our system configuration we have two chips, each consuming 275mW in the *active* state and 75mW in the *standby* state.

We consider the impact of such memory on the effectiveness of DVS for our MPEG decoding benchmark. We expect memory to dilute the impact of DVS on overall energy consumption. First for standard Mobile-RAM operated in *active* and *standby* states, memory energy consumption can be calculated as the sum of the energy consumed in *active* and that consumed in *standby*. Also because the number of accesses and MPEG's period of 66ms are fixed, the memory energy consumption stays roughly constant (9.93mJ) as processor speed varies.

Table 3 shows that our simulation results match our expectations. This table provides statistics on CPU power, execution time, average gap for chip 0, memory energy, CPU energy and total energy for various voltage (frequency) settings. We divide memory and CPU energy into two portions. The first portion corresponds to the energy consumed while the task is executing (the busy part of the period). The second portion (labeled "Residue") is the energy consumed during the time between the task completion and the end of the period (i.e., CPU leakage power and DRAM standby for standard memory).

Table 3. DVS with Standard and Naive Powerdown Memory

							Standard		Naive	
CPU Freq (MHz)	CPU Pwr (mW)	Exec Time (ms)	Avg Gap0 (ns)	CPU Eng (mJ)	CPU Residue (mJ)	Mem Eng (mJ)	Mem Residue (mJ)	**Total Eng** (mJ)	Mem Residue (mJ)	**Total Eng** (mJ)
50	16.5	65.18	36446.5	1.08	0.00	9.81	0.12	**11.01**	0.00	**10.89**
100	38.3	32.63	18659.8	1.25	0.00	4.93	5.01	**11.18**	0.12	**6.30**
200	99.9	16.35	9487.3	1.63	0.01	2.48	7.45	**11.58**	0.17	**4.30**
400	311.0	8.22	4786.1	2.55	0.05	1.26	8.67	**12.53**	0.20	**4.07**
600	669.3	5.50	3199.3	3.68	0.10	0.86	9.07	**13.72**	0.21	**4.86**
800	1210.1	4.15	2413.4	5.02	0.19	0.65	9.28	**15.15**	0.22	**6.09**
1000	2354.7	3.34	2201.3	7.86	0.38	0.53	9.40	**18.17**	0.22	**8.99**

From the data in Table 3 we see that for this standard memory system, the lowest energy is achieved by using the lowest CPU voltage setting. Since the memory power is constant over the entire period, the lowest energy is achieved by minimizing the CPU energy. However, while the CPU energy changes by a factor of 7, the total energy savings from lowering voltage is only 39%. These relative savings would be even lower if more DRAM chips were used (e.g., in a laptop with eight memory chips).

5 DVS and Power-Aware Memory

Power-aware memory offers the opportunity to reduce the energy consumed during idle times by placing DRAM chips into lower power states. The key problem with the

traditional memory design of the previous section in the context of DVS is that DRAM still consumes relatively high power (75mW) during the slack portion of the period.

5.1 Naive Power-Awareness

An alternative to keeping memory powered on all the time is to power down both the CPU and memory for the time between task completion and the end of the period. It is this slack time that many DVS algorithms seek to minimize by stretching the execution. This "naive" implementation enables the DVS scheduler to issue a "command" that places DRAM into the powerdown state.

Table 3 shows that this naive approach lowers overall energy consumption by dramatically reducing the memory residual energy consumption which represents the energy consumed by the DRAM in powerdown mode. The memory energy costs (the sum of the memory energy column and the memory residue for naive) are brought down into the range of CPU energy. In a sense, these two components are *balanced* in terms of energy. The effect of this is to make any power management functions of either the CPU or memory relatively important. Introducing the powerdown capability in the memory yields a 51% total energy savings without frequency scaling (i.e., comparing 8.99mJ to 18.17mJ at 1GHz) and a 68% savings coupled with the best frequency.

However, we note a dramatically different effect of DVS on total energy. At 50MHz, memory remains powered on too long and dominates total energy which equals 10.89mJ. In contrast, at 1GHz execution time does not decrease enough to offset the substantial increase in CPU power and total energy is 8.99mJ. The interesting point is that the lowest total energy consumption (4.07mJ) is achieved at 400MHz. Therefore, total energy has a u-shape as a function of processor frequency/voltage.

This result conflicts with conventional assumptions used in many DVS algorithms which have been concerned only with CPU energy. *Taking into account the energy used by memory with even minimal power management capabilities, it is no longer best to stretch execution to consume the entire period.* In fact, the lowest frequency produces the highest total energy consumption in this case. Instead, the best frequency/voltage for minimizing energy should be obtained by including memory energy in the decision.

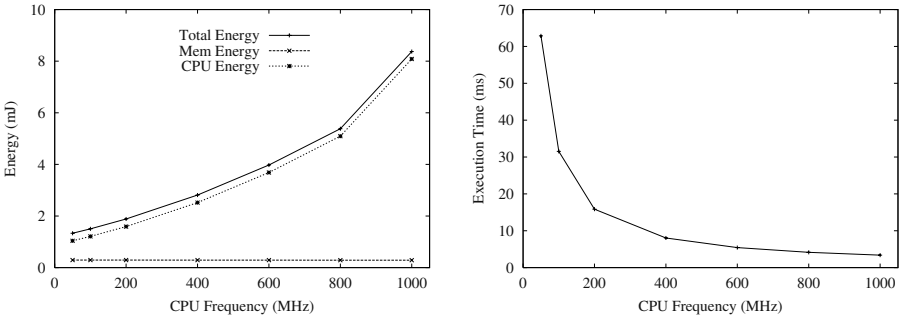
5.2 Dynamic Power-Aware Memory

Although the naive powerdown approach can reduce total energy, it does not exploit the full capabilities of power-aware memory. The low power state is entered only after task completion. Next, we investigate the interaction between processor voltage scaling and sophisticated power-aware memory that utilize low power states while a task is busy.

In contrast to the naive approach described above, this form of power-aware memory employs memory controller policies that manipulate DRAM power states during the busy portion of the task period. By default they all place the DRAM chips into powerdown for the slack portion of the task period. We begin by considering the behavior of the immediate powerdown (*aggressive*) policy for various frequency values. We note that our MPEG application fits entirely in one memory chip, thus the remaining chip can power down even while the task is busy.

Fig. 2 shows energy versus frequency (a) and execution time versus frequency (b). The three lines in the energy graph correspond to the total energy, memory energy, and

processor energy. From this graph, and the data in Table 4, we see that the *aggressive* policy has significantly different behavior than either a traditional memory system or the naive powerdown approach. At high frequency the total energy is comparable to the naive powerdown policy. However, at low frequency the total energy is much lower.



a) Energy b) Task Busy Time

Fig. 2. DVS and Power Aware Memory: MPEG Decode–Aggressive Policy

As the frequency increases from 50MHz to 1GHz, the total energy increases from 1.36mJ at 50MHz to 8.53mJ at 1GHz. These results illustrate that when aggressive memory power management is applied, CPU energy becomes dominant and traditional DVS begins to work as expected without having to exclude from consideration the energy consumption of memory. This behavior is explained by examining the processor and memory energy components. Processor energy steadily increases quadratically with the increased voltage required for each higher frequency. In contrast, the total memory energy (busy and slack portion) stabilizes at around 0.28mJ (as explained in Sec 5.4).

From the discussion thus far, we can make several important observations. First, the naive implementation that powers down memory during slack portions of the period can produce lower overall energy consumption than a standard memory. However, this

Table 4. DVS and Power Aware Memory: MPEG Decode

CPU Freq (MHz)	CPU Pwr (mW)	Exec Time (ms)	Avg Gap0 (ns)	Mem Pwr (mW)	CPU Eng (mJ)	CPU Residue (mJ)	Mem Eng (mJ)	Mem Residue (mJ)	Total Eng (mJ)
50	16.5	65.18	36398.3	4.23	1.08	0.00	0.28	0.00	1.36
100	38.3	32.63	18641.7	4.94	1.25	0.00	0.16	0.12	1.53
200	99.9	16.36	9482.3	6.35	1.63	0.01	0.10	0.17	1.92
400	310.7	8.23	4782.6	9.13	2.56	0.05	0.08	0.20	2.88
600	668.1	5.52	3203.3	11.88	3.69	0.10	0.07	0.21	4.07
800	1207.1	4.16	2496.6	14.52	5.03	0.19	0.06	0.22	5.50
1000	2347.6	3.35	2541.4	16.66	7.87	0.38	0.06	0.22	8.53

result conflicts with DVS algorithms that assume the lowest frequency will produce the lowest energy (this assumption only holds for the CPU). Fig. 3 illustrates these results by showing energy consumption versus frequency. One line is for CPU energy only, the other lines correspond to various power-aware memory policies and include both CPU and memory energy. The aggressive power management policy lowers the overall energy consumption, particularly at the lower frequencies. *A conclusion to draw from this comparison of memory policies is that more effective power-aware memory management contributes to realizing the potential of DVS.*

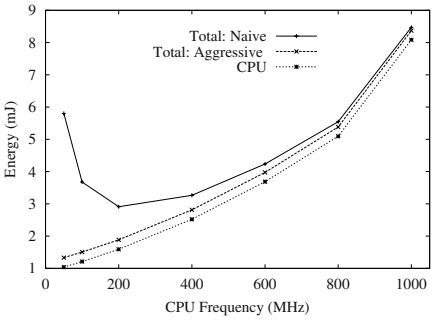


Fig. 3. DVS and Memory Controller Policies

The final observation from the above discussion concerns the influence that limitations on execution time can have on energy consumption. For MPEG this was simply the linear effects of frequency changes versus the quadratic effects on power consumption. However, other benchmarks have other execution time bottlenecks. In particular, cache behavior can have a dramatic effect on execution time for some programs. MPEG has a very low data cache miss ratio; however, several researchers have identified embedded applications that incur miss ratios from 5% to 15% depending on cache configuration. Bishop et al [2] show that PEGWIT, the public key encryption application in the MediaBench suite, has a miss ratio of 15% in a 16KB 32-way data cache. In our experiments, PEGWIT has a miss ratio of 2.3% with our 32KB 32-way cache configuration.

5.3 Miss Ratio Effects

To explore the influence of memory latency and cache performance on DVS we consider the effect of changing the workload's miss ratio on voltage setting. Since it is hard to vary the miss ratio for a fixed cache configuration with real benchmarks, we use a synthetic benchmark to create three workloads with the same 30ms period but different miss ratios: 2%, 9% and 16%. The synthetic benchmark runs 1.0–1.7 million instructions. We manipulate the instruction number and the ratio of instruction type (computation/control/memory) to generate different miss ratios while maintaining the roughly equal execution time. For each workload the 50MHz frequency is sufficient to complete task execution in the requisite period.

Our goal is to examine the behavior of each workload as the processor voltage is scaled. Therefore, we present normalized results to avoid accidental comparisons between workloads. Fig. 4a) and 4b) show the total energy normalized to the 50MHz value for the naive and aggressive policy, respectively.

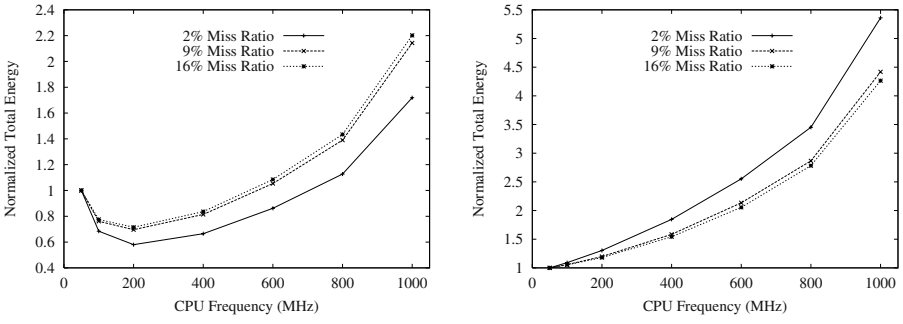


Fig. 4. Miss Ratio Effects on DVS: Normalized to the 50MHz value

From Fig. 4 we see similar trends for each workload. For the naive policy (Fig. 4a), energy initially decreases, then increases dramatically as processor power increases. For the aggressive policy (Fig. 4b), energy increases as frequency increases. We note that for the aggressive policy, the total energy increases more rapidly for lower miss ratios. This is because as miss ratio increases memory energy increases, making CPU less dominant. Also, for a given miss ratio the total memory energy remains constant as frequency varies, this additive factor makes total energy increase less rapidly for high miss ratio benchmarks than for low miss ratio ones. For the naive policy, however, we note that the overall energy energy decreases less rapidly and increases more rapidly for higher miss ratios. Firstly, because the *standby* power during execution is greater than the residue power during task slack time, memory energy does not remain constant but decreases as frequency increases, forming a trend opposite to CPU energy, thus making total energy a u-shape. Secondly, the execution time of higher miss ratio workloads are limited by memory latency sooner, making memory energy decrease less rapidly and reducing the benefit of increased clock frequencies.

These results indicate that DVS algorithms should consider memory energy consumption when setting voltage levels. Similarly, DVS algorithms should also consider memory's effect on performance when determining which frequencies meet the deadlines. The challenge is to develop a method for determining what voltage/frequency level should be used to minimize overall energy and meet deadlines. The following section outlines our approach for meeting this challenge.

5.4 Toward Memory-Aware DVS

In this section, we show how to use available information to calculate component energy (CPU and memory) and total energy for each processor frequency level.

Estimating Energy. To estimate processor energy consumption, we multiply the estimated execution time by the estimated power consumption. We use the range of CPU power values given by [16] and represent the power associated with frequency f by $P_{cpu}(f)$. To calculate the execution time, we divide it into 3 parts according to the power state of the memory: T_{act}, $T_{L\rightarrow act}$ and T_L. We use L to denote one of the low power states. It can be $standby$ for naive power aware memory or $powerdown$ for aggressive power aware memory. T_{act} is the time spent in the $active$ power state where accesses are serviced. $T_{L\rightarrow act}$ is the extra time when memory is making transitions from the low power state to the $active$ state. Fig. 1 shows each transition is followed by an $active$ duration which services at least one access. To compute these two time values, we need to know how many times the memory makes a power transition and, for each transition, how many accesses (cache misses) are serviced. We assume only one access is serviced each time and hence the number of power transitions equals the number of cache misses. We claim it is a reasonable approximation for our inorder processor model. Furthermore, our simulation results agree with this approximation. T_L is the sum of all durations when the CPU does not generate cache misses and the memory remains in the low power state. So each instruction, except those that trigger a cache miss, contributes a cycle to T_L.

From the above discussion and assuming a base CPI of one, we can calculate the execution time, T_{exec}, as follows:

$$T_{act} = t_{access} N_{misses} \tag{1}$$

$$T_{L\rightarrow act} = t_{L\rightarrow act} N_{misses} \tag{2}$$

$$T_L = \frac{1}{f}(N_{insts} - N_{misses}) \tag{3}$$

$$T_{exec} = T_{act} + T_{L\rightarrow act} + T_L \tag{4}$$

The residual time is easily computed by subtracting our estimated execution time from the provided period ($T_{residual} = T_{period} - T_{exec}$). Therefore the CPU, memory and total energy can be calculated as follows:

$$E_{cpu} = T_{exec} P_{cpu}(f) + T_{residue} P_{leakage} \tag{5}$$

$$E_{mem} = T_{act} P_{act} + T_L P_L + T_{L\rightarrow act} P_{L\rightarrow act} + T_{residual} P_{powerdown} \tag{6}$$

$$E_{total} = E_{cpu} + E_{mem} \tag{7}$$

Note all parameters required to solve the above equations are either available from the hardware specifications (t_{access}, $t_{L\rightarrow act}$, P_{act}, P_L, $P_{L\rightarrow act}$, $P_{powerdown}$, $P_{cpu}(f)$, $P_{leakage}$) or easily obtained with existing performance counters on most modern processors (N_{misses}, N_{insts}, f).

Evaluation. We use both synthetic and real workloads to evaluate our energy estimates. Fig. 5 shows the measured energy (Simulation) and our predicted energy (Predicted) versus clock frequency for PEGWIT.

The first observation is that our prediction of each component's energy and total energy matches the general shape of the simulation results. Our model works very well on memory energy prediction. We note that the errors in CPU and total energy

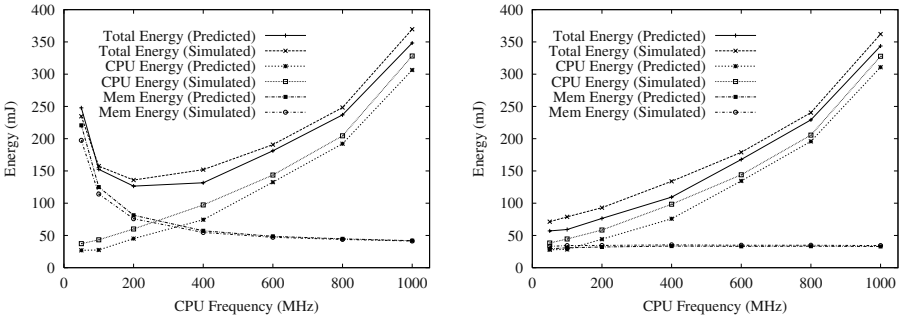


Fig. 5. Energy Prediction – Inorder Processor

estimation are primarily due to the fact that the fixed CPU power values from [16] can not accurately reflect the actual power consumption obtained from the simulation. Nonetheless, the estimates appear to be sufficient for a DVS algorithm to choose an appropriate frequency.

We also examined how our model and simulation compare for an out-of-order processor, and we get very similar results to the inorder processor. Since the out-of-order processor tends to generate multiple outstanding cache misses, equations (1,2,4) generally overpredict the execution time and thus lead to a slightly higher energy estimation for the out-of-order processor than for an inorder processor, leaving a side-effect of being more accurate (as illustrated in Fig. 6).

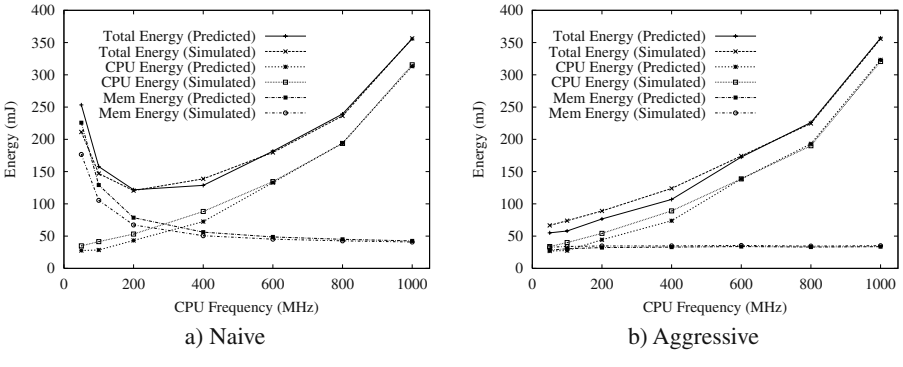


Fig. 6. Energy Prediction – Out-of-order Processor

6 Summary and Conclusions

This work shows there is a synergistic effect between dynamic voltage scaling (DVS) of the processor and power-aware memory control. Our simulation results for four appli-

cations from the MediaBench benchmark suite show that combining DVS with power-aware memory achieves greater energy savings than either technique in isolation – a consistent 89% savings compared to our standard base case. By contrast, the energy savings from DVS alone with standard memory are only 39%. Using a power-aware memory policy that transitions into powerdown mode, but without exploiting the DVS capabilities of the processor, yields a total energy saving of 54%. The interaction between these two technologies has the greatest impact.

Traditional DVS works under the assumption that CPU power dominates. Unfortunately, in a common mobile system with standard memory and low power processor, memory power is comparable to CPU power, and thus dilutes the benefit from DVS. For applications with predictable periodicity, naive power management can be used to reduce task slack time energy. However, the memory energy scales with frequency and voltage differently from the CPU energy. Therefore the tradeoff between memory and processor energy has to be considered when setting the correct speed to minimize total energy. Aggressive power management further reduces memory energy so that CPU becomes dominant again. It makes the benefit from DVS more pronounced and the best speed setting becomes compatible with DVS algorithms.

Recognizing the tradeoff between memory and processor power consumption and memory's influence on execution time, we propose a technique to estimate execution time and the total energy consumption of a given task for a given power-aware memory policy. Our approach requires information that is easily obtained with existing performance counters on many modern processors. We show that our execution time and energy estimates are sufficient to capture the tradeoff between memory and processor energy consumption, and can be used by a DVS algorithm to select an appropriate voltage/frequency setting.

As a natural extension to the above work, we also explored the impact of DVS on the memory access behavior and hence the selection of memory control policies. Due to the space limitation, we only give our conclusion here. If the DVS algorithm has to increase the frequency (i.e. to meet a deadline), the memory controller policy should adapt to the change of access behavior to gain maximum benefit (i.e. switch from an aggressive transition policy to a moderate one due to shortened gaps).

References

1. The SimpleScalar-Arm Power Modeling Project. //www.eecs.umich.edu/ jringenb/power/.
2. B. Bishop, T. Kelliher, and M. Irwin. A detailed analysis of mediabench. In *1999 IEEE Workshop on Signal Processing Systems*, November 1999.
3. V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramiam, and M.J. Irwin. DRAM energy management using software and hardware directed power mode control. In *7th Int'l Symp. on High Performance Computer Architecture*, January 2001.
4. X. Fan, C. S. Ellis, and A. R. Lebeck. Memory controller policies for DRAM power management. In *International Symp. on Low Power Electronics and Design (ISLPED)*, pages 129–134, August 2001.
5. X. Fan, C. S. Ellis, and A. R. Lebeck. Modeling of DRAM power control policies using deterministic and stochastic petri nets. In *Workshop on Power Aware Computing Systems*, February 2002.

6. K. Flautner and T. Mudge. Vertigo: Automatic performance setting for Linux. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2002.
7. K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. In *7th Annual International Conference on Mobile Computing and Networking*, pages 260–271, 2001.
8. K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *1st Annual International Conference on Mobile Computing and Networking*, November 1995.
9. D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Symp. on Operating Systems Design and Implementation (OSDI)*, October 2000.
10. C. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *International Symp. on Low Power Electronics and Design (ISLPED)*, pages 275–278, August 2001.
11. C. Hughes, J. Srinivasan, and S. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *34th International Symp. on Microarchitecture*, December 2001.
12. Infineon. *Mobile-RAM*, 2001. http://www.infineon.com/.
13. C. M. Krishna and Y-H. Lee. Voltage-clock-scaling techniques for low power in hard real-time systems. In *IEEE Real-Time Technology and Applications Symp.*, May 2000.
14. A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 105–116, November 2000.
15. C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *30th International Symp. on Microarchitecture (MICRO 30)*, December 1997.
16. S. Leibson. XScale (StrongARM-2) Muscles In. *Microprocessor Report*, September 2000.
17. J. Lorch and A. J. Smith. Operating system modifications for task-based speed and voltage scheduling. In *1st International Conf. on Mobile Systems, Applications, and Services*, May 2003.
18. J. R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with pace. In *Joint International Conference on Measurement and Modeling of Computer Systems*, 2001.
19. T. Martin. Balancing batteries, power and performance: system issues in cpu speed-setting for mobile computing. In *PhD thesis, Carnegie Mellon University*, 1999.
20. T. Martin and D. Siewiorek. Non-ideal Battery and Main Memory Effects on CPU Speed-Setting for Low Power. *Transactions on Very Large Scale Integrated Systems, Special Issue on Low Power Electronics and Design*, 9(1):29–34, February 2001.
21. T. L. Martin, D. P. Siewiorek, and J. M. Warren. A cpu speed-setting policy that accounts for nonideal memory and battery properties. In *39th Power Sources Conf.*, June 2000.
22. D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power (COLP'00)*, October 2000.
23. T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *International Symp. on Low Power Electronics and Design (ISLPED)*, 2000.
24. Trevor Pering, Thomas D. Burd, and Robert W. Brodersen. The simulation and evaluation of dynamic scaling algorithms. In *International Symp. on Low Power Electronics and Design (ISLPED)*, August 1998.
25. P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th Symp. on Operating Systems Principles*, October 2001.

26. J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *The Seventh Annual International Conference on Mobile Computing and Networking 2001*, pages 251–259, 2001.
27. V. Swaminathan and K. Chakrabarty. Real-time task scheduling for energy-aware embedded systems. In *IEEE Real-Time Systems Symp. (Work-in-Progress Session)*, November 2000.
28. M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *1st Symp. on Operating Systems Design and Implementation (OSDI)*, November 1994.

Hot-and-Cold: Using Criticality in the Design of Energy-Efficient Caches

Rajeev Balasubramonian[1], Viji Srinivasan[2], Sandhya Dwarkadas[3], and Alper Buyuktosunoglu[2]

[1] School of Computing, University of Utah
[2] IBM T.J. Watson Research Center
[3] Department of Computer Science, University of Rochester

Abstract. As technology scales and processor speeds improve, power has become a first-order design constraint in all aspects of processor design. In this paper, we explore the use of criticality metrics to reduce dynamic and leakage energy within data caches. We leverage the ability to predict whether an access is in the application's critical path to partition the accesses into multiple streams. Accesses in the critical path are serviced by a high-performance (hot) cache bank. Accesses not in the critical path are serviced by a lower energy (and lower performance (cold)) cache bank. The resulting organization is a physically banked cache with different levels of energy consumption and performance in each bank. Our results demonstrate that such a classification of instructions and data across two streams can be achieved with high accuracy. Each additional cycle in the cold cache access time slows performance down by only 0.8%. However, such a partition can increase contention for cache banks and entail non-negligible hardware overhead. While prior research has effectively employed criticality metrics to reduce power in arithmetic units, our analysis shows that the success of these techniques are limited when applied to data caches.

1 Introduction

Technology improvements resulting in increased chip density have forced power and energy consumption to be first-order design constraints in all aspects of processor design. Furthermore, in current processors a large fraction of chip area is dedicated to cache/memory structures and with each technology generation this fraction continues to grow. As a result, caches account for a significant fraction of overall chip energy. For example, in the Alpha 21264 [10], caches account for 16% of energy consumed.

In this work we focus on reducing both dynamic *and* leakage energy of L1 data caches by exploiting information on instruction criticality. Instructions of a program have data, control, and resource dependences among them. Chains of dependent instructions that determine a program's execution time are referred to as the critical paths. In other words, instructions that can be delayed for one or more cycles without affecting program completion time are considered to not be on the critical path. Such instructions, referred to as non-critical instructions, afford some degree of latency tolerance. By identifying these instructions consistently and correctly, they are directed to access a statically designed low energy and lower performance (*cold*) cache bank. Critical instructions are directed

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 180–195, 2004.

toward a (*hot*) cache bank designed for high performance. The resulting organization is a physically banked cache with different levels of energy consumption and performance in each bank.

The challenges in such an implementation are two-fold: (i) determining the bank in which to place a given piece of data, and (ii) partitioning the instruction stream into a critical and a non-critical stream. Our analysis of the data and instruction stream of a range of applications shows that the criticality of both instructions and data shows high consistency. Based on this analysis, we steer instructions to cache banks based on the instruction's program counter, and place data in these banks based on the percentage of critical instructions accessing the data line.

If the *cold* cache is designed to be highly energy-efficient (consuming 20% of the dynamic and leakage energy of the *hot* cache), we observe L1 data cache energy savings of 37%. Our results indicate that critical instruction (and data) prediction is reliable enough that performance degrades by only 0.8% for each additional cycle in the *cold* cache access time. This allows us to employ power-saving techniques within the *cold* cache, that might have dramatically degraded performance if employed in a conventional cache. However, the re-organization of data across the *hot* and *cold* banks increases contention and this degrades performance by 2.7% compared to a conventional word-interleaved cache. Hence, for the *hot-and-cold* organization to be effective, the latency cost of employing the power-saving techniques in the conventional cache has to be prohibitive. While prior work has effectively employed criticality metrics to design low-power arithmetic units [21], our results show that criticality-directed low power designs are not highly effective in L1 caches of high-performance processors.

In Section 2, we elaborate on techniques that have been proposed to address energy consumption in caches, and motivate the use of statically designed caches. In Section 3, we analyze the consistency of a program's instructions and data in terms of criticality to determine if their behavior lends itself to criticality-based classification. Section 4 describes our cache implementation. We present its performance and energy characteristics in Section 5. Finally, we conclude in Section 6.

2 Energy-Delay Trade-Offs in SRAM Design

A number of circuit-level and architectural techniques can be employed to reduce dynamic and leakage energy in caches. At the circuit level, as process technology improves, careful transistor sizing can be used to reduce overall capacitance, and hence dynamic energy. Since dynamic energy is roughly proportional to the square of V_{dd}, lowering V_{dd} can help reduce dynamic energy. Simultaneous to the above circuit-level techniques, architectural techniques such as banking, serial tag and data access, and way prediction [19], help lower dynamic energy by reducing the number of transistors switched on each access. This comes at the cost of increased latency and/or complexity. For example, delay is roughly inversely proportional to V_{dd}.

Several techniques have been proposed to reduce leakage energy while minimizing performance loss [3, 9, 11, 12, 15, 18, 28]. For example, higher V_t devices help reduce leakage energy [18]. However, when applied statically to the entire cache, especially the L1 cache, these techniques increase access latency. Since an L1 cache access time

must typically match processor speed, an increase in access latency by even a cycle can have a significant impact on performance [11]. Circuit-level techniques that use dynamic deactivation to switch to a low leakage mode increase manufacturing cost and/or affect latency and energy consumption of the fast mode (either in steady-state or due to the transition). Our goal in this paper is to examine ways by which static low-power designs of caches might be exploited using architectural techniques to reduce *both* leakage and dynamic energy consumption while minimally affecting performance.

3 Instruction and Data Classification

From the discussion in the previous section, it is clear that the cost of decreasing energy consumption of L1 caches is an increase in average access time of the cache. One way to mitigate this performance loss is to overlap the extra access time penalty incurred due to the energy saving techniques with the execution of other instructions in the program. Such overlapping is only possible if the corresponding load instruction is not on the application critical path, where the critical path is defined by the longest chain of dependent instructions whose execution determines application completion time. We propose a technique to identify such non-critical instructions and steer data accessed by these instructions to a low energy and lower performance *cold* bank, while critical loads are still served from a fast, *hot* bank.

3.1 Criticality Metrics

Recent work [7, 8, 24, 25, 26] has examined the detection of instruction (and/or data) criticality. Srinivasan and Lebeck [25] used a simulator with roll-back capabilities to accurately classify each instruction as critical or not. More recent studies have proposed heuristics that approximate the above detailed classification method to allow feasible implementations. Srinivasan *et al.* [24] and Fisk and Bahar [8] determine that load instructions are critical if they incur a cache miss, or lead to a mispredicted branch, or slow down the issue rate while waiting for data to arrive. Fields *et al.* [7] classify an instruction as critical if it is part of a chain of successive wake-up events. Tune *et al.* [26] propose a number of heuristics that predict whether instructions are critical or not. For example, their analysis shows that treating the oldest instructions in the issue queue as critical performs as well as other more complicated metrics that use data dependence chain information to determine criticality.

Our analysis also confirms that using the position of the instruction in the issue queue to determine its criticality performs comparably to techniques that use more complicated metrics. Using this *Oldest-*N technique, an instruction is deemed critical if it is among the oldest N instructions in the issue queue at issue time, where N is a pre-defined parameter. Since ready instructions that are further downstream (not among the oldest N) have a greater degree of latency tolerance, it is fair to mark them as non-critical. Note that such a heuristic tends to identify instructions along mispredicted paths as being non-critical, because mispredicted instructions are usually not the oldest instructions in the issue queue. The following advantages motivate the use of *Oldest-*N in our design: (i) No hardware table is required to predict instructions as critical because the position in

the issue queue at the time of issue is sufficient to determine criticality. (ii) The ratio of critical to non-critical instructions can be tuned by varying N.

3.2 Classifying Load Instructions

This subsection attempts to quantify the consistency of criticality behavior of load instructions with the *Oldest-N* metric. Similar results were seen when employing other complex criticality metrics. In our analysis, $N = 5$ allows the most accurate classification of instructions as critical or not. Figure 1 (a) shows a histogram of the percentage of loads that show the same criticality behavior as their last dynamic invocation.

In Figure 1(a), we observe that for most of the applications, over 85% of dynamic loads have the same criticality as their previous invocation. Only *gap* (80%) and *twolf* (84%) have a slightly lower degree of consistency. On average, over 88% of loads show consistent criticality behavior. The high consistency exhibited by loads motivates the use of hardware predictors to partition accesses into critical and non-critical streams.

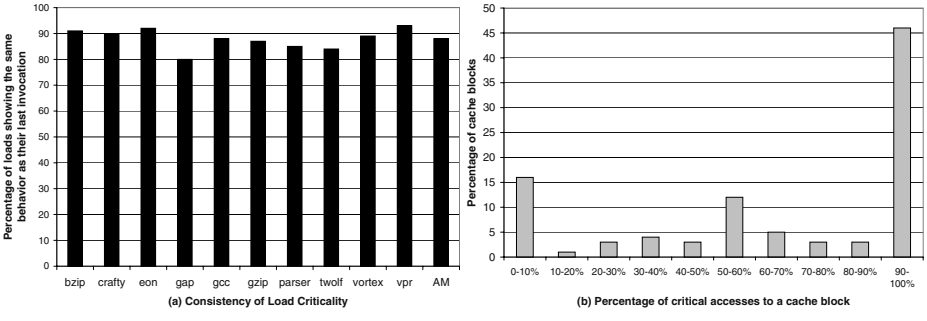


Fig. 1. (a) Consistency of load criticality. (b) Percentage of critical accesses to a data cache block – the figure is a histogram showing the percentage of data blocks that had a particular fraction of accesses from critical loads

3.3 Classifying Data Blocks

While the results in the previous subsection reveal that instructions can be statically categorized as critical or non-critical, the same behavior need not hold true for accessed data blocks. A single cache line is accessed by a number of loads and stores, not all of which may have the same criticality behavior. In order to be able to place data in either the hot or cold cache bank, we have to determine if the accesses to cache blocks are dominated by either critical or non-critical loads/stores. Figure 1(b) shows a histogram of the distribution of data cache blocks based on the percentage of accesses to each block that were attributable to critical loads. This is computed by averaging the histograms for each individual program. From Figure 1(b), we observe that nearly 46% of data cache blocks have 90 to 100% of their accesses from critical loads. Similarly, 16% of data blocks have only 0 to 10% of their accesses from critical loads (i.e., over 90% of their accesses are due to non-critical memory operations). These results indicate that

data blocks are also often exclusively critical or non-critical. However, there is a non-trivial percentage of blocks that are accessed equally by critical and non-critical loads, and steering such data blocks to one of the cache banks will impact performance and/or energy. When using smaller cache line sizes, blocks are more strongly polarized as being critical or non-critical. Although smaller lines reduce interference from access behaviors of other words in a line, there still remain a number of words that are accessed equally by critical and non-critical loads/stores.

4 The Hot-and-Cold Banked Cache

Proposed Organization. Motivated by the results in Section 3, we propose a new organization for the L1 data cache that is composed of multiple cache banks, with some banks being energy-efficient and the rest being designed for the fastest possible access time. Figure 2 shows a high-level block diagram of the proposed *hot-and-cold* L1 data cache. The L1 data cache is split into two banks – a "hot" bank and a "cold" bank. The hot bank is slated to contain data blocks that are marked critical, while the cold bank is slated to contain data blocks that are marked non-critical.

The *hot* cache provides the fastest possible access time, and the *cold* cache services requests in an energy-efficient manner while incurring a longer latency. Since the *cold* bank is similar to other conventional cache banks, it could use any proposed architectural energy-saving techniques like serial tag-data lookup, or way prediction [19] to reduce energy per access. In addition, the cold cache can be designed with more energy-efficient circuits using transistor sizing (as discussed in section 2) to reduce overall capacitance, high $\mathrm{V_t}$ for reduced leakage, or gated-ground SRAM cells [3].

Data Placement using Placement Predictor. Every time a cache block is fetched from the L2 cache, it is placed into one of the hot or cold banks based on the history of accesses to that block when it last resided in the L1 cache. For this purpose, we track the fraction of accesses to each cache block due to critical loads/stores. As explained in the last section, we use the *Oldest-N* metric to determine whether a load instruction is critical or not. For each data block in the L1 cache we maintain an n-bit up/down counter, initialized to 2^{n-1} to track the number of accesses to that block due to critical memory access instructions. The counter is incremented for every critical access to the block and decremented for every non-critical access. A 4-bit counter was chosen to avoid miscategorization of a cache line as either critical or non-critical.

When a data block is being replaced from the L1 cache, we mark the corresponding line as "critical" if the counter is greater than or equal to 2^{n-1}. Else, it is marked "non-critical". To save this information, we could use 1 extra bit (the "criticality" bit) per line in the L2 cache directory. Subsequently, when the data block is brought back into the L1 cache from L2, it is placed in the hot cache if the criticality bit of that line in the L2 cache is set to 1; otherwise, it is placed in the cold cache. The criticality bit of all lines in L2 are initialized to 1. As the criticality bit in an L2 cache line stores the most recent classification of the cache block, the placement of blocks in the hot and cold cache adapts dynamically depending on the change in access patterns to a cache block. Moreover, updates to the criticality bit are not in the critical path because they are performed only when a block is replaced from the L1 cache.

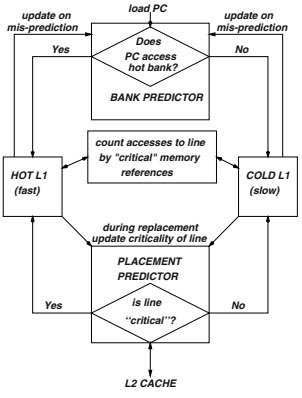


Fig. 2. The Energy-Aware Hot-and-Cold Banked Cache

In spite of the low overhead for storing this 1 bit in the L2 cache, it is desirable to eliminate it for the following reasons: (i) The bit has to be communicated to L2 even if the block being replaced is not dirty. (ii) Despite having a bit per L2 cache line, the bit is lost if the cache block is evicted from L2. Hence, we explored the use of a separate structure (the *placement* predictor) that stores these bits for a subset of cache blocks. Every time a block is evicted from L1, the block address is used to index into this structure and update the classification of that block using the criticality information for the line. The bit is set to 1 if the line is critical, and to 0 if the line is non-critical. Every time a block is brought in from L2, the structure is indexed to determine the classification and place data in either the *hot* or the *cold* bank. We found that a structure of size as small as 4K bits was enough to accurately classify blocks as being critical and non-critical. Using a separate structure avoids having the size of the table grow in proportion to the L2.

Load/Store Steering Using Bank Predictor. The above mechanism partitions data blocks across two streams. Next, we have to partition memory operations and steer them to the appropriate cache bank. For each load/store instruction, we use a predictor indexed by the instruction's PC to steer the access to the bank that contains the data being accessed. Note that this steering does not take into account the criticality nature of the instruction itself – because a critical load can access a block that is classified as non-critical and a non-critical instruction can access a block that is classified as critical.

We maintain a hardware-based dynamically updated bank predictor that keeps track of the bank that was last accessed by a particular instruction. We experimentally observed that a bank predictor of size 4Kx1 bit was sufficient to steer memory accesses with an average accuracy of greater than 90%. Moreover, a PC-based predictor to steer accesses to hot and cold banks allows the steering to be done as soon as a load/store instruction is decoded. Hence, selecting between banks does not incur any additional cycle time penalty. The predictor contains a bit for each entry. If the value of the bit is one, the access is steered to the hot cache, and if the value is zero, the access is steered to the cold cache. The counter value is set to one if the data is found in the hot cache and reset to 0 if it is found in the cold cache.

During every access, tags for both banks are accessed simultaneously. This allows us to detect a steering misprediction. For each such misprediction, we incur additional

performance and energy penalty for probing both the hot and the cold cache array. This need not introduce additional complexities in the issue logic as the operation is similar to the load replay operation on a cache miss. The variable latency across different loads can also be handled – at the time of issue, the bank being accessed (and hence, its latency) is known, and wake-up operations can accordingly be scheduled.

Related Work. The *hot-and-cold* cache has a read bandwidth of two with a single read port per bank because in any given cycle one hot and one cold word can be accessed. Thus, it behaves like a two-banked cache in which the bank steering is done based on the criticality nature of cache blocks. We therefore compare the performance of the *hot-and-cold* cache with a word-interleaved two-banked cache. Rivers *et al.* [14] show that partitioning the cache in a word-interleaved manner minimizes bank conflicts because most applications have an even distribution of accesses to odd and even words.

Recent work by Abella and Gonzalez [2] examines a split cache organization with a fast and slow cache. Contrary to our proposal, their policies for data placement and instruction steering are both based on the criticality nature of the accessing instruction. There is also considerable performance-centric related work that has looked at cache partitioning. For example, the MRU [23], Victim [13], NTS [20], and Assist [16] caches, and the Stack Value File [17] study various ways to partition the cache to improve average cache access latency. The focus of our work is quite different; the main motivation for the design choices we explored in this paper is to provide faster access to a subset of cache lines while saving energy when accessing the rest of the lines.

5 Results

5.1 Methodology

To evaluate our design, we use a simulator based on Simplescalar-3.0 [4] for the Alpha AXP instruction set. The register update unit (RUU) is decomposed into integer and floating point issue queues, physical register files, and reorder buffer (ROB). The memory hierarchy is modeled in detail, including word-interleaved access, bus and port contention, and writeback buffers. The important simulation parameters are summarized in Table 1.

Table 1. Simplescalar Simulation Parameters

Fetch queue size	16	Issue queue size	20 (int and fp, each)
Branch predictor	comb. of bimodal and 2-level	Register file size	80 (int and fp, each)
Bimodal pred. size	2048	Re-order Buffer	80
Level 1 predictor	1024 entries, history 10	Int ALUs/mult-div	4/2
Level 2 predictor	4096 entries	FP ALUs/mult-div	4/2
BTB size	2048 sets, 2-way	L1 I and D cache	16KB 2-way, 2 cycles
Branch mpred penalty	at least 12 cycles	L2 unified cache	2MB 8-way, 16 cycles
Fetch width	4	TLB	128 entries, 8KB page
Dispatch/commit	4	Memory latency	90 cycles

We estimated power for different cache organizations using CACTI-3.0 [22] at 0.1μm technology. CACTI uses an analytical model to estimate delay and power for tag and data paths. We obtained an energy per read access of 0.67 nJ for a two-banked, 16KB, 2-way associative L1 cache with a 32 byte line size. Of this, 0.56 nJ was due to bit-lines and sense amplifiers. For write accesses the cache essentially behaves like a 1-way cache, and 50% of bit-line and sense amplifier energy can be eliminated. Hence, energy per write access is 0.39 nJ ($0.67 - 0.5 * 0.56$). We also take into account the energy cost of reading and writing entire cache lines during writeback and fetch. Note that our evaluations only show energy consumed within the L1 data cache and not overall processor energy. The contribution of the data cache to total processor power can be as little as 5% in a high-performance processor and as much as 50% in an embedded processor. In future technologies, increased leakage energy is likely to increase the contribution of the L1D to total chip power. Given the wide range of possible values, we restrict ourselves to presenting the savings in data cache energy only.

We analyzed additional energy expended due to the hardware counters and predictors used in the *hot-and-cold* cache. With CACTI, we derived energy per access of the tag array for the L1 and L2 cache to be 0.055 nJ and 0.162 nJ, respectively. Based on this, we derived the energy per access due to the 4-bit counter used in the L1 tag array (for tracking critical/non-critical accesses of a cache block) to be 0.004 nJ (which is only an additional 1% energy per access for the L1 cache). Similarly, we estimated the additional energy per access to the *placement* predictor to be 0.5 pJ, which is a negligible fraction per L2 access (since the predictor is updated and accessed only on an L1 miss or eviction). The energy for the *bank* predictor used for steering memory accesses to the hot or cold bank is 0.5 pJ per access (same size as the *placement* predictor), which is again a negligible fraction of L1 data cache energy per access.

We simulated 10 programs from SPEC2k-Int[1] with reference input datasets. The simulation was fast-forwarded 2 billion instructions, another 1 million instructions were used to warm up processor state, and the next 500M instructions were simulated in detail. Table 2 presents the benchmarks with their base IPC and L1 and L2 miss rates.

5.2 Comparison of Performance

Figure 3 compares performance of the hot-and-cold cache to the baseline L1 data cache, which is dual banked and word interleaved. The first bar presents IPC of the baseline L1 data cache. The second bar shows IPC for the hot-and-cold cache with data allocation to banks based on criticality, assuming perfect steering (no bank prediction) of loads/stores to banks. The criticality metric used is *Oldest-N*, where N is fixed at 7.

By using the *hot-and-cold* organization, allocation of data across the two banks is different. As a result, overall IPC is reduced by about 1.8%, which is a result of two factors: (i) There is an imbalance in the number of accesses to each bank while using the hot-and-cold cache. Some applications (*bzip, gzip, twolf*) have many more critical accesses, while others (*gcc, vortex*) have many more non-critical accesses. This results in excess contention for the limited cache ports as compared to the word-interleaved banked

[1] *Perlbmk* did not run with our simulator and *mcf* is too memory bound for its performance to be affected by changes to the CPU and cache.

Table 2. Base Statistics for Benchmarks Used

Benchmark	Base IPC	L1 miss rate	L2 miss rate	Benchmark	Base IPC	L1 miss rate	L2 miss rate
bzip	1.42	2.96	5.22	gzip	1.37	2.84	1.37
crafty	1.32	4.37	0.16	parser	1.20	4.77	4.83
eon	1.60	0.90	0.04	twolf	1.09	9.51	0.11
gap	1.68	0.62	17.12	vortex	1.14	2.71	0.92
gcc	1.18	9.98	0.24	vpr	1.02	4.38	10.65

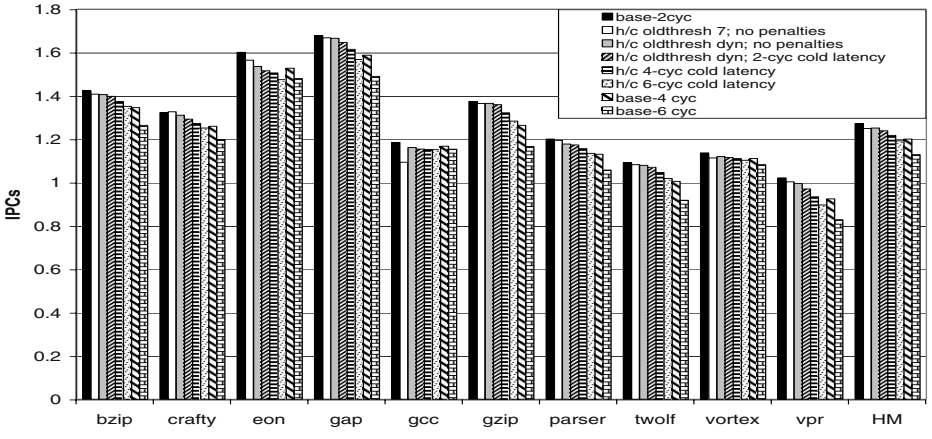


Fig. 3. Performance of Hot-and-Cold Cache Relative to Baseline Word-Interleaved

cache, in which, accesses to each bank are roughly equal. (ii) Accesses to a critical or non-critical cache are more bursty. When an instruction completes and wakes up its dependents, there is a high probability that the dependents would either all be critical or all be non-critical. Since an entire cache line resides in one bank, spatial and temporal locality also dictate that the same bank would be repeatedly accessed. We verified this by examining the number of accesses to each bank in a 10-cycle window. Figure 4 shows a histogram indicating the percentage of such windows that encountered a particular ratio of accesses to the two banks (using an *Oldest-7* threshold for the *hot-and-cold* cache). For the word-interleaved cache, an average of 36% of all time windows had roughly the same number of accesses to each bank. For the *hot-and-cold* cache, this number was only 19%, while the number of windows that had exclusively either critical or non-critical accesses was as high as 26%. This shows that reorganizing data in the banks based on criticality results in an increase in data cache port contention.

While we cannot completely eliminate the bursty nature of critical and non-critical accesses since this is inherent to program behavior, we used the following method to improve the distribution of accesses to the banks. By varying the parameter N, we changed the number of critical accesses and thus, the allocation of blocks to the two banks. We found that keeping the number of accesses to each bank roughly equal resulted in the

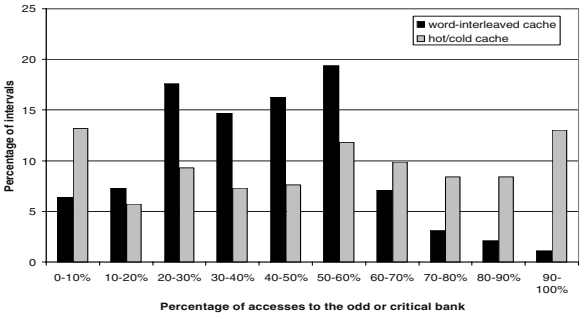


Fig. 4. Histogram showing the percentage of 10-cycle intervals that experienced a particular ratio of accesses to the two banks (using *Oldest-7* for the *hot-and-cold* cache)

least amount of port contention. This is achieved at run-time with a simple mechanism that uses statistics over a past interval to determine the value of N for the next interval. We used 10M instruction intervals and discarded statistics from the first half of an interval. Over the latter half of the interval, we counted the number of accesses to each bank. In terms of hardware, this requires two counters to keep track of the number of accesses to the two banks as well as a comparison triggered every 10M instructions. If the percentage of accesses to the hot bank was less than 45%, we increased the value of N so as to classify more loads as critical. Likewise, if the percentage of accesses to the hot bank was more than 55%, we decreased N. The statistics from the first half of each 10M instruction interval are discarded to allow enough cache block evictions and fetches that the parameter change is reflected in statistics collected for the latter half. Striving for a 50% share of critical accesses minimizes port contention, improving IPC by 1%. However, we found that keeping the share of critical accesses to 60% was better at minimizing IPC loss from a slower cold cache. The third bar in Figure 3 represents such an organization and is only marginally better than a fixed value of $N = 7$. However, since it classifies more instructions as critical, it sees a much lower performance degradation when cold cache latency is increased. Note that this represents a model where loads and stores are perfectly steered to the bank that caches the data. *Gcc* is an example of a program that is highly constrained by cache port contention. Using a value of $N = 7$ causes a high imbalance in accesses to each bank and degrades performance. As a result, the dynamic tuning of N is very important in this case to minimize additional port contention stalls causes by the *hot-and-cold* cache organization.

The fourth bar in the figure shows a model that uses the *bank* predictor. An incorrect prediction results in a probe of both the banks, resulting in a higher latency and greater port contention. Due to this, overall IPC goes down by an additional 1%. We found that the mispredict rate was 9.5% on average, thus confirming our earlier hypothesis about the easy predictability of the nature of blocks accessed by loads and stores.

Table 3 shows various statistics that help us explain the changes in performance. The table shows that the hot-and-cold cache organization has to handle more accesses and this increase is caused by mispredictions while steering loads and stores. We also note that the distribution of accesses to odd and even banks in the base case is fairly even (except in *gcc*). By tuning the value of N, the distribution of accesses to hot and

Table 3. Data access statistics for the base word-interleaved cache and the *hot-and-cold* using a dynamic *Oldest-N* threshold

benchmark	base case accesses	bnk1:bnk2 accesses	port conflict stalls	H-and-C accesses	Hot:Cold accesses	port conflict stalls	steering mpreds
bzip	191M	60:40	41.6M	198M	62:38	103.6M	7.1M
crafty	194M	58:42	59.1M	216M	59:41	127.1M	21.1M
eon	232M	56:44	76.8M	263M	59:41	231.3M	33.0M
gap	183M	57:43	51.6M	213M	60:40	106.0M	30.2M
gcc	345M	76:24	1754M	354M	53:47	1894M	8.7M
gzip	177M	62:38	80.0M	185M	58:42	117.8M	7.9M
parser	175M	62:38	62.6M	187M	59:41	106.1M	12.3M
twolf	173M	58:42	74.1M	194M	62:38	94.4M	24.6M
vortex	218M	63:37	176.4M	241M	36:64	267.7M	24.3M
vpr	210M	51:49	59.1M	238M	58:42	134.5M	29.8M

cold banks is adjusted to approximately be in the ratio 60:40. The notable exception is *vortex*, where most instructions issue from the last issue queue entry and are classified as non-critical. Because of the increased number of accesses to the *hot-and-cold* cache and the inherent bursty nature of these accesses (Fig 4), we see that the number of stall cycles due to port contention is much higher. On average, the hot-and-cold cache has twice as many stall cycles as the word-interleaved base case (In *gcc*, the program is already highly constrained by port contention, so the increase caused by the *hot-and-cold* re-organization does not result in a doubling of the number of stall cycles.).

Finally, the fifth and sixth bars show the effect of increasing the cold cache latency to four and six cycles, respectively. The seventh and eighth bars show IPCs for a word-interleaved cache where all accesses take four and six cycles, respectively. In spite of slowing down as many as 45% of all memory operations, increasing cache latency from 2 to 4 cycles only results in a 1.6% IPC loss. Uniformly increasing the latency of every access to 4 cycles in the base case results in an IPC penalty of 5.7%. Thus, the use of the criticality metric helps restrict the IPC penalty of a slower cache access time. However, owing to the penalty from increased port contention and mis-steers, the hot-and-cold cache with the 4-cycle cold cache latency does only marginally better than the 4-cycle word-interleaved cache. The value of the proposed organization is seen when the energy-saving techniques threaten to slow the cache to a latency of six cycles. The hot-and-cold organization with the 6-cycle cold cache latency outperforms the 6-cycle base case by an overall 5.7%, demonstrating its ability to tolerate the additional latency.

As we demonstrated in Figure 1, there are a number of blocks that are not easily classifiable as critical or non-critical. This causes some amount of inefficiency in the system – when critical loads access non-critical blocks, performance is lost, and when non-critical operations access critical blocks, they unnecessarily consume additional energy. We noticed that 26% of all memory operations were of this kind. This inefficiency cannot be microarchitecturally eliminated as it is an artifact of the program that the same data is accessed by different kinds of loads and stores.

Finally, it could be argued that a word-interleaved cache like the base case with half the total capacity could match the energy consumption of the *hot-and-cold* cache if the base capacity is not fully utilized. We make the assumption (which we verified for our base case design parameters) that the capacity of the base case is chosen to work well across most programs and that halving its capacity would severely impact a number of programs, rendering such an organization unattractive.

5.3 Comparison of Energy

Energy-Delay Trade-Offs. Our proposal does not make any assumption on the particular energy-saving techniques that can be employed for the cold bank. Since the cold bank is like any other conventional cache bank, it could use any of the already proposed architecture or circuit level power-saving techniques. Hence, our results are parameterized across multiple access time and energy consumption characteristics.

In order to provide a more detailed understanding of energy-delay trade-offs possible through circuit-level tuning, we performed circuit simulations using a typical SRAM cross-section from the predecoder to the wordline driver in 0.13μ CMOS technology. The predecoder consists of 3-to-8 NORs and the decoder is an n-input NAND, where n is the number of 3-to-8 predecode blocks. Finally, the wordline drivers consist of inverters that drive the load, which includes all associated wires and SRAM cells. We used a formal static tuning tool, EinsTuner [5], to vary total device width. EinsTuner [27, 5] is built on top of a static transistor-level timing tool (EinsTLT) that combines a fast event-driven simulator (SPECS) with a timing tool (Einstimer). The SPECS simulator provides timing information such as delay and slew along with first derivatives with respect to transistor width. EinsTuner uses this information to formulate the optimization problem as a linear combination of slack and area. This formulation is then solved by a non-linear optimization package LANCELOT [6], which treats all device widths as free parameters and solves for minimum delay. Finally, energy values are obtained using AS/X circuit simulations [1] for a given switching activity.

Figure 5 shows normalized energy-delay trade-offs for an SRAM cross-section. The primary y-axis shows delay (*norm.delay*) corresponding to a given energy consumption.

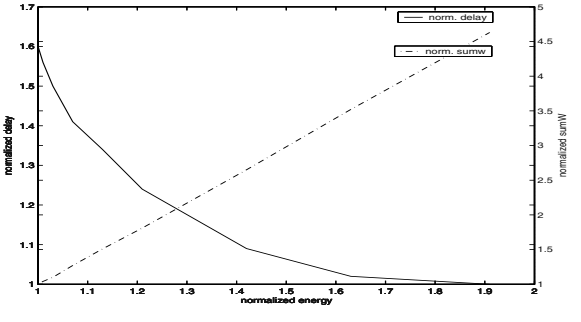


Fig. 5. Energy-Delay Curve for SRAM Cross-section. The dotted line plots the normalized sum of transistor widths against the normalized energy consumption. The solid line plots the normalized delay against the normalized energy consumption

The secondary y-axis shows the normalized sum of transistor widths (*norm.sumw*). For this experiment, we used parameters such as beta constraint (i.e. PMOS/NMOS width ratio), internal slew rate, primary output slew rate, and input capacitance/stage constraints from real designs. The initial data point for this experiment is obtained by minimizing delay without area constraints, and the other points show minimum delay with specific area constraints. From Figure 5, we observe that by increasing minimum delay by 60%, we can achieve up to 48% reduction in energy.

Table 4. Leakage Energy-Delay Trade-offs for Different V_t

V_t	Normalized Leakage Energy	Normalized Delay
low	8.5	0.88
nominal	1	1
high	0.23	1.34

Table 4 shows normalized leakage energy-delay trade-offs for the same SRAM cross-section using transistor widths that correspond to minimum delay. We observe that decreasing V_t increases leakage energy dramatically (8 times), while increasing V_t decreases leakage energy by 77% at the expense of 34% increase in delay.

Dynamic and Leakage Energy Savings. As discussed in Section 2, many techniques can be employed to reduce cache energy, including transistor sizing, lowered V_{dd}, banking, serial tag and data access, higher V_t, etc. Since any of the above techniques can be applied to the *cold* bank, we present results for energy savings assuming the cold bank consumes either 0.2 or 0.6 times the dynamic and leakage energy consumed by the hot bank. Results for using just one of the circuit-level techniques — transistor sizing from Figure 5 — would lie in between (corresponding to roughly 0.5 time the dynamic energy consumed by the hot bank, with roughly twice the access latency in cycles). When designing with a higher V_t for the cold bank, the 77% reduction in leakage energy shown in Table 4 corresponds roughly to 0.2 times the leakage consumed within the hot bank with roughly a 1 cycle increase in delay for our cache organization. The use of multiple techniques could potentially bring more aggressive energy savings at a potentially higher access penalty, justifying our choice of range in terms of energy savings (40% to 80% of the base case) and access penalty (1.5 to 3 times the base case).

Figure 6 shows potential energy savings from the proposed organization. For each program, we show L1 data cache energy for three organizations - (i) word-interleaved base case, (ii) hot-and-cold organization, where the hot bank has characteristics identical to a bank of the base case, and the cold bank consumes 0.6 times the dynamic and leakage energy consumed by the hot bank, (iii) hot-and-cold organization, where the hot bank is the same and the cold bank consumes 0.2 times the energy consumed by the hot bank. For the hot-and-cold cache, the figure also shows the contribution to total energy from the two banks. Since 45% of all accesses are steered to the cold bank, that number serves as an approximate upper bound to the potential energy savings.

By having a highly energy-efficient cold bank (like that represented by the third bar in the figure), the energy consumption in the data cache reduces by an average of 37%.

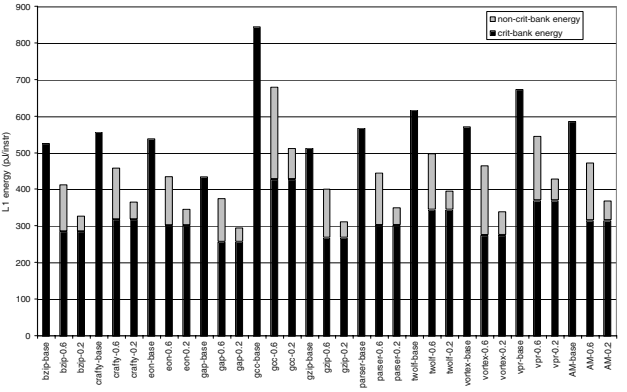


Fig. 6. L1 data cache energy (both leakage and dynamic). The first bar represents a word-interleaved base case. The second and third bars represent a hot-and-cold cache organization, where the dynamic and leakage energy within the cold bank are 0.6 and 0.2 times the energy within the hot bank, respectively. For the hot-and-cold cache, the black and grey portions of the bars represent the energy consumed within the hot and cold banks, respectively

There is little effect on L2 energy consumption since the miss rates of the two caches are comparable. Leakage energy consumed is a function of total execution time (which is slightly longer for the *hot-and-cold* cache). We observed that the contribution to total energy savings came equally from dynamic and leakage components.

Note that energy savings can be further increased by improving steering prediction accuracy. Our results take into account the additional energy overhead of a steering misprediction – about 10% of all loads and stores access both banks. Also note that the distribution of accesses across different banks is almost the same in all programs, resulting in very little variation in energy trends across the benchmark set.

6 Conclusion

We have presented and evaluated the design of a banked cache, where each bank can be fixed at design time to be either *hot* or *cold*, i.e., high energy and low latency, or low energy and high latency, respectively. The performance impact of accessing the *cold* cache can be minimized effectively by separating load and store instruction streams into a critical and non-critical stream. Our results demonstrate that performance impact is reasonably insensitive to the latency of the *cold* bank, allowing aggressive power reduction techniques. This is made possible by the consistent classification of instructions and data as critical and non-critical streams. Each additional cycle in the *cold* cache latency impacts performance by about 0.8%. Energy savings are proportional to the fraction of accesses to the *cold* cache, with L1 energy reduction being an average of 37% (compared to a word-interleaved base case) for an energy-efficient *cold* bank.

However, allocation of data blocks in the *hot* and *cold* banks increases contention and introduces bank steering mispredicts. This results in an IPC degradation of 2.7%, compared to a word-interleaved conventional cache, that severely limits the effectiveness

of this approach. To alleviate these problems, bank prediction would have to be improved or base cases with low contention would have to be considered. Such problems were not encountered in the design of criticality-based arithmetic units with different power/performance characteristics [21]. The *hot-and-cold* cache becomes more effective when the cost of employing any power-saving technique becomes prohibitive. For example, a *hot-and-cold* organization with a 2-cycle *hot* latency and a 6-cycle *cold* latency outperforms a 6-cycle word-interleaved base case by 5.7%.

We are working with circuit designers in order to help define the energy consumption ratio between hot and cold cache banks. Our initial analysis reveals that simple techniques like transistor sizing and high V_t can dramatically reduce dynamic and leakage energy consumption, validating the choice of parameters in our evaluation. We also plan to evaluate the use of asymmetric sizes (and organizations) for hot and cold banks.

References

1. *AS/X User's Guide, IBM Corporation, New York*. 1996.
2. J. Abella and A. Gonzalez. Power Efficient Data Cache Designs. In *Proceedings of ICCD-21*, Oct 2003.
3. A. Agarwal, H. Li, and K. Roy. DRG-cache: A data retention gated-ground cache for low power. In *Proceedings of the 39th Conference on Design Automation*, June 2002.
4. D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.
5. A. R. Conn, I. M. Elfadel, W. W. Molzen, P. R. O'Brien, P. N. Strenski, C. Visweswariah, and C. B. Whan. Gradient-based optimization of custom circuits using a static-timing formulation. In *Proceedings of Design Automation Conference*, pages 452–459, June 1999.
6. A. R. Conn, N. I. M. Gould, and P. L. Toint. *LANCELOT: A Fortran Package for Large-Scale Non-Linear Optimization (Release A)*. Springer Verlag, 1992.
7. B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of ISCA-28*, July 2001.
8. B. Fisk and I. Bahar. The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency. *IEEE International Conference on Computer Design*, October 1999.
9. K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. *29th Annual International Symposium on Computer Architecture*, May 2002.
10. M. Gowan, L. Biro, and D.Jackson. Power Considerations in the Design of the alpha 21264 Microprocessor. In *35th Design Authomation Conference*, pages 726–731, June 1998.
11. H. Hanson, M. S. Hrishikesh, V. Agarwal, S. W. Keckler, and D. Burger. Static Energy Reduction Techniques for Microprocessor Caches. *2001 International Conference on Computer Design*, September 2001.
12. S. Heo, K. Barr, M. Hampton, and K. Asanovic. Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines. *29th Annual International Symposium of Computer Architecture*, May 2002.
13. N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA-17)*, pages 364–373, May 1990.
14. J.Rivers, G. S. Tyson, E. Davidson, and T. Austin. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, Dec. 1997.

15. S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. *28th Annual International Symposium on Computer Architecture*, June 2001.
16. G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W. Bryg. Pa7200: A pa-risc processor with integrated high performance mp bus interface. *COMPCON Digest of Papers*, pages 375–382, 1994.
17. H. Lee, M. Smelyanskiy, C. Newburn, and G. S. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 5–14, Jan. 2001.
18. K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano. A low power sram using auto-backgate-controlled MT-CMOS. In *International Symposium on Low-Power Electronics and Design*, 1998.
19. M. Powell, A. Agrawal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via selective direct-mapping and way prediction . *34th Annual International Symposium on Microarchitecture*, December 2001.
20. J. A. Rivers and E. S. Davidson. Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 151–162, Aug. 1996.
21. J. Seng, E. Tune, D. Tullsen, and G. Cai. Reducing Processor Power with Critical Path Prediction. In *Proceedings of MICRO-34*, Dec 2001.
22. P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report TN-2001/2, Compaq Western Research Laboratory, August 2001.
23. K. So and R. Rechtschaffen. Cache operations by mru change. *IBM Technical Report, RC-11613*, 1985.
24. S. T. Srinivasan, R. D. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.
25. S. T. Srinivasan and A. R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. *Journal of Instruction-Level Parallelism*, 1, Oct 1999.
26. E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of HPCA-7*, Jan 2001.
27. C. Visweswariah and A. R. Conn. Formulation of static circuit optimization with reduced size, degeneracy and redundancy by timing graph manipulation. In *IEEE International Conference on Computer-Aided Design*, pages 244–251, November 1999.
28. S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High Performance I-Caches. *Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

PARROT: Power Awareness Through Selective Dynamically Optimized Traces

Roni Rosner, Yoav Almog, Micha Moffie, Naftali Schwartz, and Avi Mendelson

Microprocessor Research, Intel Labs, Haifa, Israel
{roni.rosner, yoav.almog, micha.moffie, naftali.schwartz, avi.mendelson}@intel.com

Abstract. We present the PARROT concept aimed at both higher performance and power-awareness. The PARROT microarchitectural framework integrates trace caching, dynamic optimizations and pipeline decoupling. We employ a gradual and selective approach for applying complex mechanisms only for the most frequently used traces to maximize the performance gain at any given power constraint, thus attaining finer control of tradeoffs between performance and power awareness.

We show that the PARROT microarchitecture delivers performance increases comparable to those available through conventional doubling of execution resources (average 16% IPC improvement). This improvement comes through better utilization of all available resources with the combination of a trace cache and selective trace optimization. On the other hand, performance advantage of a trace cache alone is limited to wide-machine configurations. No less critical, however, is power awareness. The PARROT microarchitecture delivers the performance increase at a comparable energy level, whereas the conventional path to higher performance consumes an average 70% more energy. Meanwhile, for those designs which can tolerate a higher power budget, PARROT gracefully scales up to use additional execution resources in a uniformly efficient manner. In particular, a PARROT-style doubly-wide machine delivers an average 45% IPC improvement while actually improving the Cubic-MIPS-per-WATT power awareness metric by over 50%.

1 Introduction

Revolutionary and evolutionary advances in microarchitecture and process technology have sustained Moore's law—doubling both transistor density and processor performance on average every 18 months—against all odds. However, despite the decreasing power consumption per individual transistor characteristic of newer process technologies, innovations in microarchitectural techniques and process technology increases the total number of transistors and their operation frequency, resulting in an overall increase in power consumption as well as power density. Doubling cache sizes, adding new and more complex types of speculation and other modern innovations require so many more resources that their own benefit is compromised. Current processors have become power limited, that is, they can only operate at limited frequency, preventing them from achieving their full microarchitectural performance potential.

B. Falsafi and T.N. Vijaykumar (Eds.): PACS 2003, LNCS 3164, pp. 196–214, 2004.

In this paper we focus on power-aware microarchitectures for high-performance, general-purpose processors. The essential challenge within this domain is the increasingly poor scaling of performance with power consumption. Nevertheless, the concepts and methods we present can be applied to special purpose, low-power and low-performance processor types as well.

Traditionally, processor microarchitecture can be divided into front-end and back-end parts. The front-end supplies the instructions to be executed in program order and the back-end executes them, possibly out-of-order, and commits the executed instructions (in-order again) updating the architectural state. The front-end bandwidth is crucial for the overall performance of the system — it depends on the number of instructions that can be predicted, fetched, decoded and renamed in parallel. Since decoding variable-length CISC instructions, such as those characteristic of the IA32 architecture, is an essentially serial activity, huge decoders must be employed to introduce the needed parallelism. Such decoders consume vast quantities of power; incremental enlargement brings diminishing incremental performance returns.

The back-end presents different power/performance tradeoffs. Although execution bandwidth scales with the number of execution units, instruction scheduling efficacy does not linearly scale with instruction window width. The power and complexity of dynamic scheduling depends on execution bandwidth as well as on program behavior and the instruction window size [15][3]. This is one of the main reasons that low-power architectures tend to spurn dynamic scheduling, preferring instead (VLIW like) static scheduling as a means for reducing the scheduling power [21].

This paper presents a different approach for microprocessor design that addresses various challenges posed by both the fetch and scheduling stages. Following Amdhal's Law, we propose to take advantage of the time-honored hot/cold (or 90/10) paradigm. The hot/cold paradigm asserts that a "small" portion of the ***static*** program code is responsible for "most" of its ***dynamic*** execution. This paradigm is utilized in the context of profiling compilers, dynamic translators and other program-manipulating systems — the small portion of ***frequent*** (or ***hot***) code becomes the focus and the target of most optimization efforts.

Identifying frequently executed code sections for optimization has been applied in the software-based schemes reported in [16][7][1][10]. More recently, similar methods were suggested for hardware-based systems [18][19][23][22][8][26][30]. The various proposals d+iffer in the methodology and resources used for detecting the hot paths, the structure and address space used for storing them, and the timing and resources used for optimization.

The PARROT concept aims to aggressively exploit the hot/cold paradigm in hardware for the benefit of both processor performance and power awareness, as indicated by the PARROT acronym: Power-Aware aRchitecture Running Optimized Traces. The current study examines several microarchitectural alternatives based on this concept (we refer to them as PARROT microarchitectures). These microarchitectures are organized around an optimized trace cache. Trace caches [24][28] were mainly proposed for obtaining higher fetch bandwidth by capturing and reusing the dynamic flow of instructions irrespective of program order [20][25]. In [26] it was observed that trace-cache based mechanisms are also useful for reducing power consumption, but that differing characteristics of traces in a system may enhance either

power or performance. In the current work we study the limits of simultaneous achievement of both performance and power awareness.

The PARROT microarchitecture is designed to effectively identify the most frequent sequences of program code, aggressively optimize them once, and then efficiently execute them many times. Trace selection and filtering identify the hot code, a dynamic optimizer optimizes it, and a trace cache stores traces for repeated execution. Gradual constructions of traces, pipeline decoupling, and specific trace optimizations are key factors for power awareness. Limiting the hardware dedicated to the cold part of the code may exact a small price in performance. In return, more aggressive hardware may be used to improve performance/power tradeoffs for the dominant hot segments of the code. Indeed, our results show that no additional amortized energy is spent for the optimization of hot traces.

The simulation framework provides both performance and energy measurements in order to establish performance and power/performance tradeoffs. We consider two "reference" microarchitectural models: a standard modern processor (a 4-wide, superscalar, super-pipelined, out-of-order microarchitecture) and a brute-force straightforward 8-wide extension. We examine several new power-aware high-performance alternatives, including trace-cache based PARROT extensions of the reference models, and dynamic hot-trace optimization within the trace-cache models. Our results show that PARROT microarchitectures indeed attain both high performance and power awareness by efficiently exploiting the machine's available resources.

The rePlay system [22], although targeting performance issues, has much in common with PARROT techniques. PARROT and rePlay share the dual front-end, the decoupled, post-retirement construction of traces, and dynamic optimization of traces stored in a trace-cache. To promote power awareness, PARROT proposes a finer decoupling of trace construction based on gradual filtering in order to improve controllability of competing design metrics. PARROT's trace construction criteria are mostly static, enabling better adaptability to program structure. A good example is the handling of loops: by cutting loops at iteration boundaries, the PARROT microarchitecture prevents redundancy in the trace cache while still allowing loop unrolling. In contrast, the dynamic selection criteria of rePlay are in better synergy with the prediction mechanism. Our results complement and strengthen the rePlay study [30] showing the significant contribution of dynamic optimizations to IA32-based processors.

PARROT indeed goes beyond rePlay optimization scope by introducing core-specific optimizations which heavily exploit the fact that the optimizations are integrated into the hardware (their particular contribution is quantified in a separate paper [1]).

The dichotomy between regular and irregular code led Turboscalar [4] into the design of separate pipelines, a deep-and-narrow pipeline for the irregular code, and a shallow-and-wide pipeline for regular code (an alternative, power-oriented dichotomy is proposed in [6]). Turboscalar execution optimizes the resource bandwidth required for higher performance. PARROT builds upon a similar conceptual separation of pipelines but allows for alternative implementations of the concept. The PARROT framework is used to study performance and energy trade-offs of several alternative structures and organization of the processor pipeline over a wide range of benchmarks.

The rest of the paper is organized as follows. Section 2 describes the PARROT concept and microarchitecture in detail. Section 3 describes the simulation framework

and defines the microarchitectural models compared in the current study. Section 4 presents the simulation results, and finally, Section 5 concludes with a summary and ideas for future studies.

2 PARROT Microarchitectural Framework

In this section we study various microarchitectures based on the PARROT concept. Performance and energy are evaluated within the presented simulation framework.

2.1 The PARROT Concept

PARROT is based on the following observations:

- The working set of a program at any given time is relatively small.
- Much of the complexity excesses of modern OOO processors result from handling rare "corner cases".
- Small segments of code which are repeatedly executed ("hot-traces") typically cover most of the program's working set at any given time.
- The hot segments of code behave differently from the rest of the code, namely they are more regular and predictable, and consequently they exhibit higher potential for ILP extraction than the other, less frequently executed parts of the code.

The PARROT concept suggests basing the development of high performance power-aware systems on an ***asymmetric decoupling*** of the processor pipelines; a slightly different decoupling concept is proposed in [4]. Fig. 2.1 presents the conceptual structure of a decoupled PARROT system. The left-hand part is responsible for executing the cold portion of the code and the right-hand side executes the hot portion of the code.

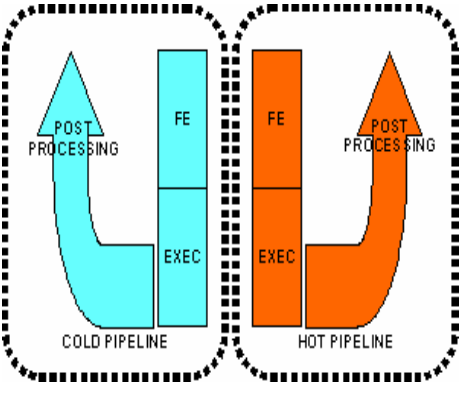


Fig. 2.1. Schematic PARROT split-core µarch

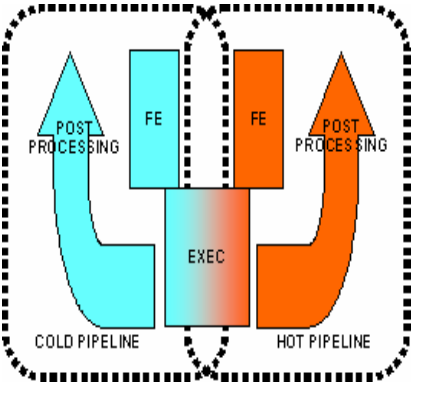


Fig. 2.2. Schematic PARROT unified-core µarch

The cold and hot subsystems have a similar high level structure, with each being comprised of foreground and background components operating in parallel. The foreground components include the front-end and execution pipelines which are tuned for either cold or hot portions of the code. The background components post-process the instruction flow out of the foreground pipeline making "off critical path" decisions such as when to move from the cold subsystem to the hot subsystem and when to apply further optimizations. Synchronization elements are required for arbitrating and switching states between the pipelines and for preserving global program order (architectural state).

Previous research [26][14] indicates that a trace cache can be very efficient in handling hot code, provided this code has been sufficiently well identified. (This is especially true for Intel's IA32 ISA which features variable length instructions.) Thus, we base the cold subsystem on instructions fetched from an instruction cache whereas the hot subsystem is based on traces fetched from a trace cache. Both power-awareness and trace-cache effectiveness considerations limit trace construction and trace-cache insertion to frequently executed code sections. Thus, PARROT ***gradually*** applies dynamic optimizations — the hotter the trace is the more aggressive power aware optimizations are applied.

Reusability of hardware work and results is important for both performance and energy savings. In the PARROT framework, the trace-cache is the container for reusable work. The trace cache stores decoded traces thus enabling the reuse of decoding results. It also stores optimized traces allowing for multiple reuses of the optimizations.

Dynamic optimizations have several advantages. Dynamic information available at optimization time, most notably control resolution (outcome of trace internal branches), enables optimizations that are impossible for a static compiler. Decoupling these optimizations from the foreground pipeline allows for more aggressive optimizations than on-the-fly optimizations that can be performed within a standard execution pipeline. In order to take full advantage of such a relaxed hardware context, ***atomicity*** of traces is assumed. Trace semantics that assumes atomic commit of traces permits for very aggressive optimizations across basic-block boundaries, including elimination and reordering of instructions, provided the overall semantics of the trace is preserved

Another advantage of microarchitectural level optimizations is that of ***architectural transparency***. The hardware is capable of optimizing legacy code, exploiting new microarchitectural features without the need for recompilation.

2.2 Traces and Trace-Selection

An execution ***trace*** is a sequence of operations representing a continuous segment of the dynamic flow (execution) of a program. Traces may contain execution beyond control-transfer instructions (CTIs), and so a trace may extend over several basic blocks. In the current study we consider ***decoded atomic traces***. These traces contain ***decoded*** micro-operations (uops) and enable reuse of decode activity, thus saving energy [29][26] (decoded traces are of special value for IA32). Traces are constructed

from the originally decoded uops in program order, but may later be optimized, resulting in an out-of-order, different, generally shorter sequence of uops.

Atomic traces are single-entry single-exit traces [17][28]. Although atomic trace semantics requires a relatively complicated recovery mechanism and longer recovery time for the case of misprediction, it enables more aggressive optimizations, including uop reordering and elimination and branch promotion [22][23] and may efficiently utilize advanced trace prediction techniques [12].

Trace selection is the process of deciding which points in the dynamic instruction stream should be designated as trace start and end points. In the current study we apply the following deterministic selection criteria:

- Trace capacity is capped at 64 uops.
- With the exception of extremely large basic blocks, traces always terminate on CTIs.
- All ***indirect jumps*** and software exceptions terminate traces, except RETURN instructions. In addition, ***taken backward*** branches also terminate a trace.
- RETURN instructions terminate traces only if they exit the outermost procedure context already encountered in the current trace.
- If two or more consecutive traces are identical, they are joined into a single trace, until the capacity limit is reached. This criterion, together with the taken-backwards termination condition on traces, achieves the effects of ***explicit loop unrolling***, an enabler for other optimizations.

With these criteria, unique ***trace identifiers*** (**TIDs**) can be compacted into a single address and a sequence of branch directions (taken/not taken). The only indirect CTI in this construction is a RETURN, but since its calling context is already part of the trace, its target address is implicitly available

2.3 Microarchitecture

A ***split-execution*** implementation, faithful to the PARROT microarchitectural concept, consists of two disjoint subsystems for the cold and for the hot paths as presented in Fig. 2.1. Consequently, different execution engines can be employed by each subsystem. For example, a wider execution engine could be used for the higher-bandwidth, trace-based hot subsystem. More sophisticated variants, not considered in the current study, may employ completely different execution models for the disjoint subsystems, such as in-order and out-of-order. An optimized ***unified-execution*** implementation is presented in Fig. 2.2 - it duplicates the front-end but shares the execution resources between the hot and cold subsystems.

The following description generally applies to both the split and unified-execution microarchitecture implementations. A schematic description of the major components is depicted in Fig. 2.3. Both cold and hot pipelines operate in two phases: the ***background*** (or ***post-processing***) ***phase***, which serves to select frequent parts of the executed code, improve (optimize) them and potentially promote them to a "hotter" level, and the ***foreground phase,*** which is responsible for the fetch-to-execution pipeline.

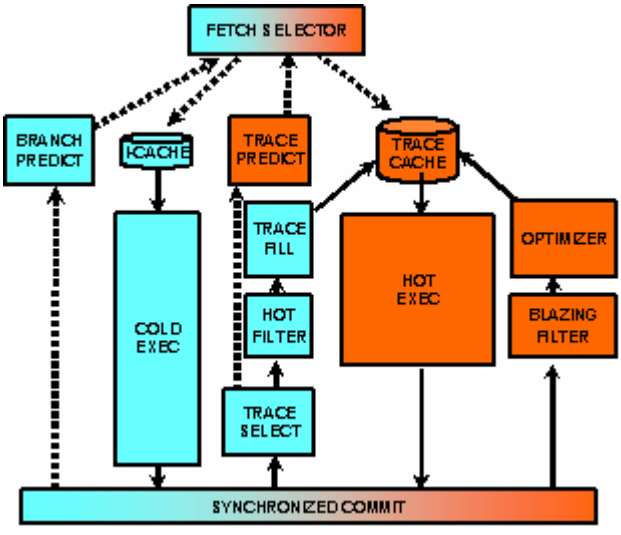


Fig. 2.3. PARROT main μarch components

The background phase of the cold subsystem identifies frequent IA32 instruction sequences and captures them as traces in the trace cache. It is composed of TID selection, TID hot-filtering and finally trace-construction and insertion into the trace-cache. Since all committed instructions enter the TID selection phase, continuous training of both trace predictor and hot filter is assured. Nevertheless, only those TIDs that pass the hot-filter continue to the trace construction stage. The background phase of the hot subsystem identifies the most frequent (blazing) traces, optimizes them and finally inserts them back into the trace cache. Post processing is gradually performed, so the longer a trace is used the more aggressive optimizations are applied to it.

Two predictors are employed: A branch predictor predicts the next cache line designated to be fetched from the instruction cache for execution on the cold pipeline. Simultaneously, a trace predictor predicts the TID of the next trace designated to be fetched from the trace cache and executed on the hot pipeline. Each predictor is based on a global history register (GHR). The GHR is updated for each executed CTI. Both predictors support speculative update upon fetch and real update upon commit.

The fetch-selector chooses between the execution pipelines by consulting both the branch predictor and the trace predictor. When the trace predictor is successful in making a next TID prediction and a trace is successfully fetched from the trace-cache, it is executed in the hot pipeline. Otherwise, cold pipeline execution is commenced using the result of the branch predictor.

In the foreground phase the pipelines are executing sequences of uops originating from either (cold) instructions or (hot) traces. This is performed by either two split cores, a cold core and a hot core, respectively, or by a single unified core. The advantages of a unified core include smaller overhead for cold/hot state switches and smaller design size (with positive impact on idle power). A split core on the other hand enables core specialization. The cold core may specialize for example on the

execution of rare but complex operations or be less performance aggressive. The hot core may specialize in aggressive execution of atomic traces, employ simplified renaming schemes or rely on dynamic scheduling performed by the optimizer. In this study we consider standard super-scalar out-of-order cores only, in both split and unified configurations.

There is some overhead associated with core switching, the magnitude of which is dependent on the number of switches and the cost of each switch. Switching cores in mid-execution requires synchronization and communication. The second core must use the correct values for registers already computed by the first core, and must not use them until they are ready. A sufficient but not necessary condition for this is to place a barrier between execution completion on one core and execution commencement on the other, enforcing ***full-serialization***. This scheme may be contrasted with the ***fine-serialization*** approach, where execution is overlapped between the cores while satisfying all dependencies between instructions executing in different cores. In this report we shall consider fine-grain serialization.

The commit stage is responsible for committing IA32 instructions to the architectural state. This stage deals with two synchronization issues: First, it has to commit instructions in program order, which means that in a split microarchitecture instructions must contain markers which permit reconstruction of the global program order from the interleaved execution. Secondly, the atomic hot traces should be committed at once as a single entity, requiring a mechanism for state accumulation. Upon any intermediate event that prevents the full completion of the trace, the remaining uops and accumulated state are flushed, and the architectural state at the commencement of the trace is restored. Such distracting events may result from exceptions in the execution of the trace itself, from failing of **assert** uops (indicating trace mispredictions) or from external interrupts.

For post-processing cold code, PARROT employs a non-speculative TID/trace build scheme. Uops originating from cold committed instructions are collected as long as all encountered CTIs satisfy the trace selection criteria (see Section 2.2). When a termination condition is reached, a new TID, generated from the collected CTIs, is used to train the trace predictor. If the TID is subsequently identified as frequent, the collected uops are used to construct an executable trace that can be inserted into the trace cache.

In order to identify the frequently executed instruction sequences, PARROT gradually employs two filtering mechanism: the hot filter, which is used for selecting frequent TIDs from among those constructed on the cold pipeline, and the blazing filter, which is used for selecting the most frequent TIDs from among those executed on the hot pipeline. Both filters are small caches that retain access counters for each TID. Each trace execution increments the corresponding counter. Once the hot filter threshold is reached, the trace is constructed and inserted into the trace cache. When the blazing filter threshold is reached, the executed trace is optimized and written back to the trace cache, replacing the original.

PARROT employs dynamic optimizations on blazing traces (identified by the blazing filter) facilitating supreme execution efficiency in performance as well as power. The optimizer heavily relies on the atomicity assumption (manifested by **assert** operations [22]), which enables aggressive trace-transformations (e.g. reordering).

2.4 Dynamic Optimizations

The optimizations can be classified as either general purpose or core-specific optimizations. General purpose optimizations are independent of the underlying execution core. They include logic simplifications, constant propagation and dead code elimination. Core-specific optimizations include functional transformations such as micro-operation fusion and SIMDification, and global trace transformations such as partial renaming and dynamic critical path based scheduling. A companion paper [1] is dedicated to a detailed study of such optimizations.

Optimizations may result in: uop reduction, dependency elimination, simplified renaming and improved scheduling. Some optimizations, such as virtual renaming, contribute mainly to power/energy saving. Others, such as dependency elimination are performance oriented. Reducing the number of micro-operations contributes in general to both performance and energy savings.

3 Simulation Framework

We employ an in-house proprietary simulation environment as a modeling and research vehicle for the PARROT microarchitecture. The simulators are designed with maximum flexibility and configurability in order to enable a comparative study of the diverse set of microarchitectural alternatives described in Section 3.3 below, based on a diverse set of benchmark applications. The simulators are trace-driven, simulating execution traces of applications compiled for the IA32 architecture. They implement all the components of the PARROT microarchitecture, including the less traditional optimizer and all of the optimizations and pre-computations described above.

3.1 Performance Simulation

The performance simulator incorporates a full memory hierarchy and newly designed components for the post-processing phases.

The software architecture includes a generic, highly configurable object-oriented execution core class which can be instantiated with a variable number of execution cores of widely differing characteristics, including machine width, number of ROB ports and number and latencies of execution units. Because it incorporates base classes suitable for executing both cold-instructions and hot-traces, it can be used to construct widely differing machine configurations.

One uncommon feature of our simulation framework is the ***abstract instruction***. An abstract instruction has a different interpretation within the cold and hot pipelines. Within the cold pipeline, it remains the familiar instruction, but within the hot pipeline, it is the trace. This design decision enabled both hot and cold execution cores to be instantiations of a generic code base. Furthermore, to maintain a high-level plug-and-play simulation semantics, the execution cluster is not a transformer of instructions to uops, as is customary. Rather, it outputs the same abstract instruction data type that it accepts for input. This design decision greatly increased the versatility of the entire system and allowed experimentation with interesting configurations which short-circuited one or both execution clusters.

We modeled the optimizer as a non-pipelined unit which stores one trace in a simplified ROB-like structure and analyzes the constituent uops sequentially, employing standard rename tables. The optimizer maintains a static dependency graph, which is used across different optimization passes. The optimizations are carried out in several passes, each pass responsible for optimizations that require similar resources. Consequently, we have modeled a significant delay (on the order of 100 cycles) for optimization of a single trace.

3.2 Energy Simulation

For energy simulation we employed a WATTCH-like approach, using accurate and up-to-date data extracted from real processors for the internal components, ensuring the relevancy of our results. Our methodology for power estimation was designed pursuant to a number of objectives. As above for performance simulation, it had to incorporate sufficient flexibility to model widely differing microarchitectures. Furthermore, it had to effectively cope with different levels of abstraction, i.e. different microarchitecture components are described in differing levels of detail. Moreover, it had to fully cover all types of design, including arrays, random logic and data paths.

The power-modeling infrastructure is based on Intel propriety tools. These include formulas for small functional blocks, each of which was closely correlated to the corresponding hardware implemented on recent technology. The formulas are composed of arithmetic expressions involving parameters, which stand for dynamic events (i.e. counters). A formula is designed to predict the dynamic energy consumption of the block, and it is composed of two sub-formulas covering both active and idle power.

The methodology we developed allows us to use the correlated formulas as building blocks, and compose them into larger formulas describing the energy consumption of higher-level units in our microarchitecture. In this composition process we utilize three mechanisms: ***instantiation, scaling*** and ***mapping***. For each unit in the microarchitectural model we instantiate a set of formulas that best fit the functionality and energetic behavior of that unit. For each formula we have to provide a mapping of its original parameters into actual event-counters of the simulated model. In addition, each formula is potentially the subject of scaling representing uniform size or activity enlargements over the original block. There are separate scaling factors for active and for idle power, representing extensions such as increased number of ports, increased (or decreased) number of entries in a cache, etc.

Consequently, power modeling for conventional units is relatively straightforward, whereas the power modeling for new units which have no similar original formulas (e.g., optimizer or trace construction), requires more work. For example, the power model of the optimizer employs instances of buffers and tables taken from the original ROB, rename and scheduling stages of the pipeline, scaled down to accommodate a single trace, and with some combinations of significant optimizer events mapped to the original execution events.

With this methodology, each microarchitectural model has its own set of formulas corresponding to the specific units being instantiated. Performance simulation runs provide the actual statistics used to compute the energy consumption for the units when their values are plugged into the corresponding formulas. The total energy

consumption of a model is the total of its constituent formulas. Approximate relative contribution per unit is the fraction of the value of a formula of this total.

Leakage is derived from the dynamic power under some simplifying assumptions. We assume uniform leakage 1) in space over two coarse component types, the processor core and the level-2 cache, and 2) in time, modeling a consistently high temperature.

To emulate the high temperature, we choose the application with highest average dynamic-power *PMAX* of the base OOO model. This turns out to be swim of the SpecFP suite (see below). For a component area *A*, we define the uniform leakage power as *A * PMAX * T*, where *T* is a technology constant. We use technology constants of 5% for each MByte of level 2 cache and 40% for the standard core. These fairly large constants are used in accordance with the technological trend of increasing leakage. Thus, for a model with *M* Mbytes of L2 cache and a core of area *K* times the standard OOO core, the total leakage energy *LE* of an application running for *CYC* cycles is modeled by the formula *LE = PMAX * (0.05 * M + 0.4 * K) * CYC*.

This infrastructure produces energy estimations for the different microarchitectural models and their components, usable for global trade-off analysis.

3.3 Models

We modeled a variety of configurations to elucidate various aspects of the power and performance of the PARROT microarchitecture. The reference model (**N**) was configured to resemble a standard 4-wide OOO machine. In the ensuing discussion, *narrow* will refer to different variants of this standard 4-wide, while *wide* should be understood to refer to variants of a more generous 8-wide. The configurations space we consider is depicted in Table 3.1.

Table 3.1. Two-dimensional configuration space

Configuration Dimensions		Traditional width	
		Narrow	Wide
PARROT accumulated features	Base	N	W
	Selective TC	TN	TW
	Optimizer	TON	TOW
	Split core	TOS	

Building on our reference base narrow configuration, we created a theoretical configuration where all stages from front end through retirement are wide (**W**). Although today's front ends cannot support 8-wide fetch (again, in particular for a variable length ISA such as IA32), the model helps in comparing the benefits of the PARROT microarchitecture to those available through incremental improvements.

The PARROT microarchitecture itself is expressed most naturally through a configuration where a narrow front end is joined to an execution engine capable of executing trace-cache based optimized hot traces. The PARROT configurations are denoted by **TON**, **TOW** and **TOS**, signifying a narrow, wide or split (narrow for cold, wide for hot) cores, respectively. The **T** stands for selective trace-cache and **O** stands

for dynamic optimizations. Finally, to assess the significance of the role trace optimizations play in improving performance, two additional configurations are included based on **N** and **W**, including selective trace-caching, but with trace optimizations disabled: **TN** and **TW**, respectively. The microarchitectural properties of all seven configurations are summed up in Table 3.2.

Table 3.2. μarch settings of different models

Parameter	N	W	TN / TON	TW / TOW	TOS
Relative core area	1	2	1.3 / 1.4	2.3 / 2.4	3.1
Predictor entries (branch + trace)	4K	8K	2K + 2K	4K + 4K	
FHR length (branch + trace)	16	16	16 + 32	16 + 32	
Icache size	64 KB	64 KB	32 KB	32 KB	
FE pipeline width (cold + hot)	4	8	4 + 4	4 + 8	
ROB entries	100	150	100	150	100 + 150
Sched. Window	32	50	32	50	32 + 50
Exec. Ports	4	8	4	8	4 + 8
EXEC pipeline width	4	8	4	8	4 + 8
Hot filter: entries, threshold	-	-	1K, 24	1K, 8	
Blazing filter: entries, threshold	-	-	1K, 32	1K, 16	
Tcache entries	-	-	128	512	
Max uops in trace	-	-	64		
Optimizer latency	-	-	100 cycles, non pipelined		
L1 Dcache: size, latency	64 KB, 3 cycles				
L2 Ucache: size, latency	2 MB, 9 cycles				
Memory latency	120 cycles				
Line size (I and D)	64 B				
All caches (I, D,T)	8-way associative, LRU				

3.4 Benchmarks

Our benchmark suite covers a wide range of application traces, 30 or 100 million instructions each. The 44 application runs can be classified as follows:

- **SpecInt 2000**: bzip, crafty, eon, gap, gcc, gzip, parser, perlbmk, twolf, vortex, vpr (30M).
- **SpecFP 2000**: ammp, apsi, art, equake, facerec, fma3d, lucas, mesa, sixtrack, swim, wupwise, (30M).
- **Office / Windows** applications from SysMark 2000: excel, office, powerpoint, virusscan, winzip, word (100M).
- **Multimedia**: flash, photoshop (from SysMark-2000, 100M), Dragon, lightwave, quakeIII, 3DsMax (light, anisotropicwheel, raster and geom), two Flask-MPEG4 runs (custom Multimedia traces, 30M).
- **DotNet**: one image, two numerical and two phong runs (100M).

3.5 Metrics

We employ a variety of metrics designated to evaluate and compare different aspects of the simulated models. For the overall processor performance we focus on the IPC, total energy and Cubic-MIPS-per-WATT (CMPW) measurements. IPC and total energy are useful for understanding the design tradeoffs assuming the same frequency and same voltage. The CMPW metric is instrumental in quantifying the design tradeoffs and power awareness of a processor assuming energy consumption could be always traded for performance using voltage or frequency scaling [5][31]. In practice, the applicability range of such tradeoffs is limited by variable technological constraints which are beyond the scope of this research.

In addition, we present some of the most important and illuminating parameters characterizing the PARROT microarchitecture, such as coverage, uop reduction and energy breakdown.

4 Results

This section examines the performance and power awareness of alternative enhancements applied to the reference 4- and 8-wide OOO machines **N** and **W**, respectively. We consider overall performance and power tradeoffs over a variety of configurations and benchmarks, as well as a detailed examination of the contribution of different components to the overall result. The **TOS** conceptual microarchitecture statistics are presented only as a reference for alternative future developments.

4.1 Performance and Power Awareness

We start by examining the impact of the PARROT extensions on the performance of N and W, respectively, as depicted in Fig. 4.1. The integration **TW** of a trace cache into the wide machine provides an additional 7% to the IPC, whereas the same extension to the narrow model (**TN**) has a negligible 2% performance benefit, mainly since the machine remains 4-wide balanced. However, the PARROT based design that includes gradual optimizations of the hot traces is much better equipped to utilize the available resources. The **TON** model achieves 17% performance improvement over **N** and the full blown **TOW** improves IPC by more than 25% over **W**. It can be observed that the irregular SpecInt applications and the execution-limited multi-media applications benefit the less from the trace-cache alone.

Considering the additional energy required for achieving the above mentioned performance improvements, we observe in Fig. 4.2 that all the extensions to the wide machine actually save energy. The reason for that is the vast energy inefficiency of the base wide machine (see Fig. 4.5 below). Regarding the narrow machines, only the TW configuration exhibits significant 12% increase in energy consumption. PARROT style optimizations result in either negligible 3% increase over **N** or a 18% energy saving over **W**.

The CMPW criterion weights both the performance gain and the energy loss as indicated in Fig. 4.3. Power awareness of the PARROT-based models **TON** and **TOW** improves over the corresponding base machines by 32% and 92%, respectively.

So far we have presented measurements relative to the base line models. It is of equal interest to consider the trade-offs between the extreme microarchitectural alternatives. We consider the base-models compared among themselves, the narrow PARROT TON compared to the brute-force wide W and finally the full-blown TOW compared to the baseline N (Fig. 4.4, Fig. 4.5 and Fig. 4.6). The major message here is the viability of the PARROT approach as a reasonable performance alternative to straightforward machine widening, with far better power awareness.

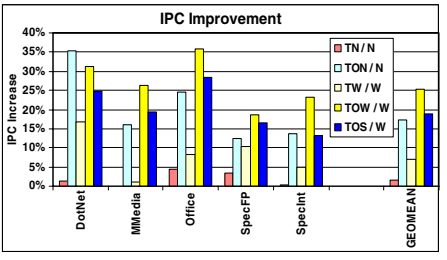


Fig. 4.1. IPC improvement over baseline of same width: N or W

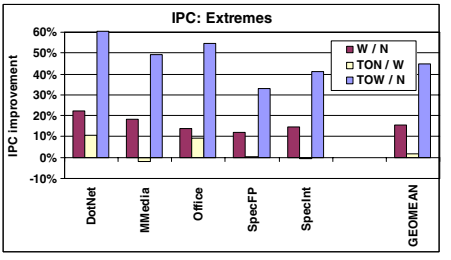


Fig. 4.4. IPC improvement across configurations

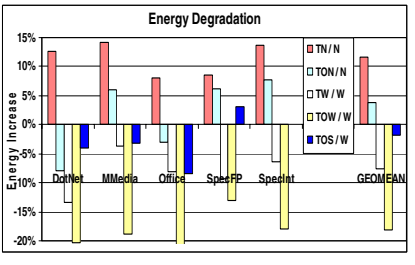


Fig. 4.2. Increased energy consumption over baseline

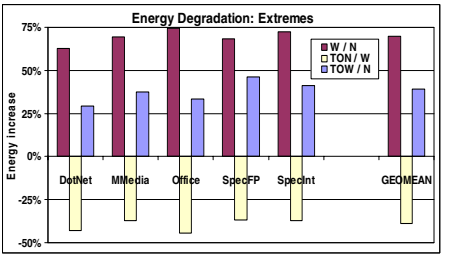


Fig. 4.5 Energy degradation across configurations

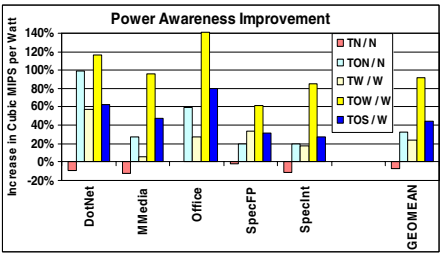


Fig. 4.3. Improved power awareness, over baseline of same width

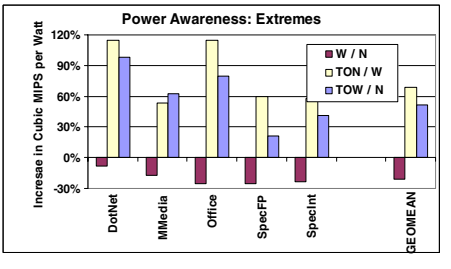


Fig. 4.6. CMPW across configurations

First, we observe that widening a machine systematically contributes to performance, but inevitably consumes more energy. The interesting phenomenon is the **TON** configuration which not only slightly outperforms the doubly wide machine **W**, but it does it with significant 39% lower energy consumption. Weighting both metrics into CMPW we can see that the PARROT extensions are 67% better than mere widening. Nevertheless, if a large power budget is available, the combination of both wider machine and PARROT-style extensions exhibited by the TOW configuration can provide an average 45% IPC increase as well as 51% CMPW improvement over base-line **N**.

4.2 Front-End Capabilities

We demonstrate the better predictability of hot code using misprediction values for the baseline **N** model with a 4K-entries branch predictor versus the PARROT **TON** model with branch and trace predictors, 2K-entries each. Fig. 4.7 shows the behavior of the PARROT machine clearly split between the hot code, for which the trace misprediction rate is even smaller than the branch misprediction rate of **N**, as opposed to the cold code, for which the branch misprediction rate is significantly the highest. This split demonstrates the better predictability of the PARROT-constructed hot traces over the cold code.

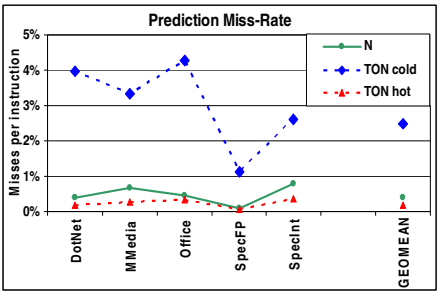


Fig. 4.7. Normalized (per instruction) misprediction rates

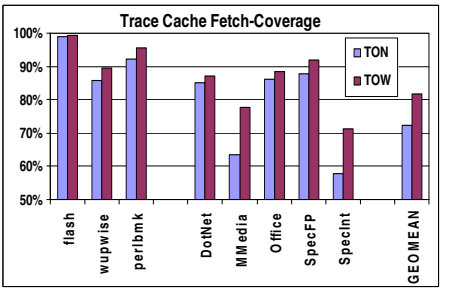


Fig. 4.8. Trace-cache hot (fetch) coverage

Coverage, as depicted in Fig. 4.8, represents the quality of the trace prediction, selection and filtering mechanisms with respect to the trace-cache size and the benchmark characteristics. The coverage is about 90% for the very regular SPEC-FP applications but around the 60-70% for the control intensive SPEC-INT applications.

The impact of a smaller trace cache of the narrow models (128 entries) vs. the wider models (512 entries) is manifested in the large coverage gaps between **TON** and **TOW** exhibited by larger working-set applications such as SPEC-INT or multimedia. Low coverage is particularly harmful for the **TN** configuration in which all the potential performance benefit is due to the trace cache‘s high fetch-bandwidth. This explains the correlation between the low coverage of SpecInt and multi-media applications on the narrow models and the fact that the **TN** model exhibits no IPC improvement for these models (see Fig. 4.1 above).

4.3 Optimizer Capabilities

The direct contribution of dynamic optimizations is most significantly reflected in the reduction in the dynamic number of executed uops and the reduction in average trace critical (dependency) path. Reducing the number of uops contributes significantly to both performance gain and reduction in active energy consumption. Shorter critical paths enable better scheduling in terms of higher ILP and shorter execution time. Overall shortening of the application execution time decreases the leakage and idle energy consumption. These contributions of the optimizer are depicted in Fig. 4.9 which shows average uop reduction of 19% and average dependency reduction of 8%. Note the relatively higher dependency reduction on the quite complex code of SpecInt.

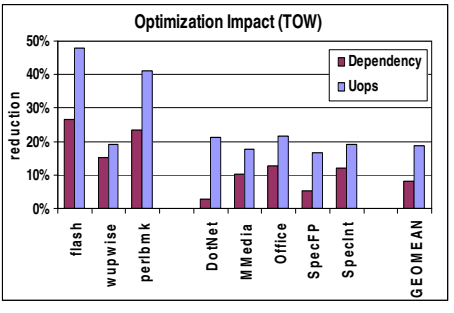


Fig. 4.9. Optimizer impact: reducing number of executed uops and simplifying dependencies

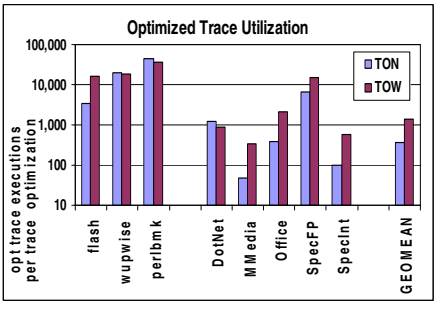


Fig. 4.10. Utilization of the optimizer work

Another important aspect of the microarchitecture is the amount of reuse of the work done by the optimizer. Fig. 4.10 depicts the optimization utilization in terms of average number of dynamic execution of optimized traces. The highest reusability is exhibited by the SPEC-FP applications due to the good spatial locality of traces in the trace cache.

4.4 Energy Breakdown

The energy breakdown between the major components of three models is depicted in Fig. 4.11. The models include the baseline **N**, the very power-aware narrow PARROT model **TON** and the conceptual, widest PARROT model **TOS**. Breakdown is shown for three applications of quite different characteristics: flash, swim and gcc.

It is interesting to note the diminishing energy contribution of the front-ends as we move from **N** to **TON** and then to **TOS**. Additionally, on a wider machine such as **TOS** the energy contribution of all execution components increases. Note that total energy required for trace manipulation, including filtering, construction of uops into atomic traces and advanced optimizations is in the order of 10% of the overall energy consumption.

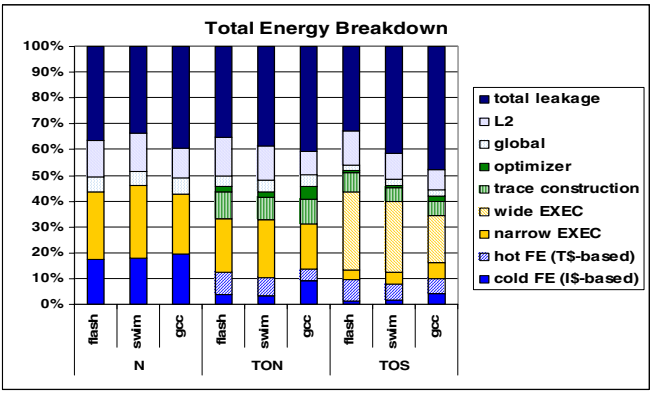


Fig. 4.11. Energy breakdown for different models

5 Conclusions and Future Work

In this paper we presented the PARROT microarchitectural framework aimed at improving both processor performance and power awareness. The PARROT concept consists of asymmetric decoupling of the processor into subsystems responsible for handling the cold (infrequent) and hot (frequent) portions of the code, thus designing each part according to different power and performance considerations. PARROT based microarchitectures use an instruction cache for the cold code and a decoded trace cache and gradual optimization techniques for executing the hot segments of the code.

We have established clear advantage of the PARROT-based approach for designing general-purpose, high-performance power-aware processors. The presented simulation results demonstrate that applying the PARROT concept to a standard, 4-wide, OOO processor yields comparable performance to an 8-wide processor, however, consuming significantly less energy. Applying the PARROT concept to the wider processor achieves significant improvement on both performance and energy.

One major topic for future research is related to split-core microarchitectures. We intend to investigate the potential advantage of such design for establishing even better performance/energy tradeoffs by considering different alternatives for the decoupled split cores.

Acknowledgements

We would like to thank Sanjay Patel, Ronny Ronen and Yiannakis Sazeides for stimulating discussions, encouragement and contributions to earlier research. We thank Lev Finkelstein and Efi Rotem for their contribution to the power modeling.

References

[1] Y. Almog, R. Rosner, N. Schwartz and A. Schmorak, "Specialized Dynamic Optimizations for High-Performance Energy-Efficient Mi-croarchitecture", to appear in *CGO'04*.

[2] V. Bala, E. Duesterwald and S. Banerjia, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo", TR HPL-1999-78, HP Labs.

[3] M. Bekerman, A. Mendelson and G Sheaffer, "Performance and Hardware Complexity Tradeoffs in Designing Multithreaded Architectures", in *PACT*, pp 24-34, Oct. 1996.

[4] B. Black and J.P. Shen, "Turboscalar: A High Frequency High IPC Microarchitecture", in *ISCA27*, June 2000.

[5] D.M. Brooks et al, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors", *IEEE Micro*, 20(6):36-44, Nov./Dec. 2000.

[6] G. Cai, C.H. Lim and W.R. Daasch, "Thermal-Scheduling For Ultra Low Power Mobile Microprocessor", in. *WCED'02*, 2002.

[7] K. Ebcioglu and E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", in *ISCA24*, pp. 26-37, 1997.

[8] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel and S.S. Lumetta, "Permormance Characterization of a Hardware Mechanism for Dynamic Optimization", *MICRO34*, 2001.

[9] D. Friendly, S. Patel and Y. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", in *MICRO31*, Nov. 1998.

[10] M. Gschwind, E.R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, "Dynamic and Transparent Binary Translation", in *IEEE Computer Magazine* 33(3), pp. 54-59, 2000.

[11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel, "The Microarchitecture of the Pentium® 4 Processor", in *Intel Technology Journal*, 2001.

[12] Q. Jacobson, E. Rotenberg and J.E. Smith, "Path-Based Next Trace Prediction", in *MICRO30*, 1997.

[13] S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, and R. Ronen, "eXtended Block Cache", in *HPCA6*, Jan. 2000.

[14] O. Kosyakovsky, A. Mendelson and A. Kolodny, "The Use of Profile-based Trace Classification for Improving the Power and Performance of Trace Cache Systems", in *4th Workshop on Feedback-Directed and Dynamic Optimization*, Austin, Dec. 2001.

[15] M.S. Lam and R.P. Wilson, "Limits of Control Flow on Parallelism", in *Proc. 19th ISCA*, pp. 46 -57, May 1992.

[16] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, "Effective Compiler Support for Predicated Execution using the Hyperblock", in *MICRO25*, 1992.

[17] S. Melvin and Y Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA", in *Intern. Journal of Parallel Prog.*, 23(3) pp 221-243, Jun. 1995

[18] M.C. Merten, A.R. Trick, C.N. George, J. Gyllenhaal, and W.W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization", *ISCA26*, 1999.

[19] M.C. Merten, A.R. Trick, E. M. Nystrom, R.D. Barnes and W. Mwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots", in *ISCA27*, May 2000.

[20] R. Nair and M.E. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups", in *Proc. ISCA24*, pp. 13-25, 1997.

[21] A. Parikh, M. Kandemir, N. Vijaykrishnan and M.J. Irwin, "VLIW Scheduling for Energy and Performance" in *Proc. IEEE Workshop on VLIW*, pp. 111-117. April 2001.

[22] S. Patel and S. Lumetta, "rePlay: A Hardware Framework for Dynamic Optimization", in *IEEE Trans. on Computers*, 50(6), pp 590-608, June 2001

[23] S. Patel, T. Tung, S Bose and M. Crum, "Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions", in *MICRO33*, 2000.

[24] A. Peleg and U. Weiser, "Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line", U.S. Patent 5,381,533, Jan. 1995.

[25] M. Postiff, G. Tyson and T. Mudge, "Performance Limits of Trace Caches", in *Journal of ILP,* vol. 1, Oct. 1999.

[26] R. Rosner, A. Mendelson and R. Ronen, "Filtering Techniques to Improve Trace-Cache Efficiency", in *PACT'01*, Sept. 2001.

[27] R. Rosner, M. Moffie, Y. Sazeides and R. Ronen, "Selecting Long Atomic Traces for High Coverage", in ICS'03, pp. 2-11, 2003.

[28] E. Rotenberg, S. Bennett and J. Smith, "A trace cache microarchitecture and evaluation", in *IEEE Trans. on Computers*, 48(2), pp 111–120, Feb. 1999

[29] B. Solomon, R. Ronen, D. Orenstien, Y. Almog and A. Mendelson "Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA", in *ISLPED'01*, Aug. 2001.

[30] B. Slechta et al, "Dynamic Optimizations of Micro-Operations", in HPCA9, Feb. 2003.

[31] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski and P.G. Emma, "Optimizing Pipelines for Power and Performance", in *MICRO 35*, 2002.

Author Index