11

Parallel Program Development and Environments

This chapter describes software environments and program development techniques for parallel computers. We first introduce environments, synchronization, and execution modes. Then we describe methods for shared-variable and message-passing program development. The emphasis is on program modularity, fast communication, load balancing, and performance tuning.

11.1 PARALLEL PROGRAMMING ENVIRONMENTS

An environment for parallel programming consists of hardware platforms, languages supported, OS and software tools, and application packages. The hardware platforms vary from shared-memory, message-passing, vector processing, and SIMD to dataflow computers. Usually, we directly identify the machine models used, such as Cray Y-MP, BBN Butterfly, iPSC/860, etc.

The last two decades have seen a revolution in massively parallel computer architecture, with system performance reaching hundreds of teraflops and even petaflops. Huge advances in processors, memory, display systems, system interconnects, and networking have contributed to this revolution, while the range of applications of such systems has also grown enormously. Newer applications of such systems include multimedia applications, data mining, and highly sophisticated simulations in science and engineering. As system hardware has developed rapidly, the accompanying parallel programming environments have also evolved and become more powerful; attempts have also been made to develop newer paradigms for parallel programming. While the basic concepts of parallel program development will be studied in this chapter, we shall review some of the more recent advances in Chapter 13.

11.1.1 Software Tools and Environments

Parallel programming languages such as Linda and Strand 88 provided the minimal parallel programming environment. Others form an integrated environment consisting of an editor, a debugger, performance monitors, and a program visualizer for improving software productivity and the quality of application programs, such as the packages Express and TOPSYS.

Figure 11.1 shows a classification of environment types on the line between the minimal languages and integrated environments. Integrated environments can be divided into *basic*, *limited*, and *well-developed* classes, depending on the maturity of the tool sets.

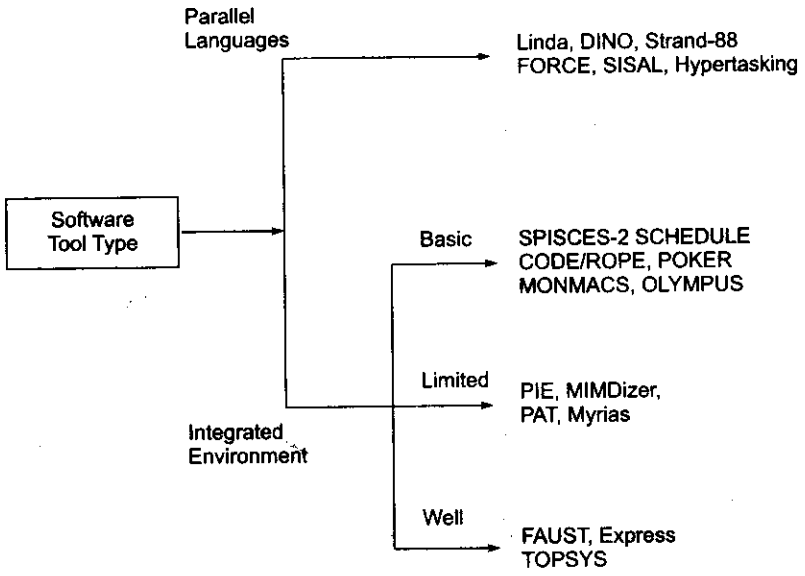


Fig. 11.1 Software tool types for parallel programming (Courtesy of Chang and Smith, 1990)

A basic environment provides a simple program tracing facility for debugging and performance monitoring or a graphic mechanism for specifying the task dependence graph in SCHEDULE, the process call graph in FAUST, and the process component graph in PIE.

Limited integration provides tools for parallel debugging, performance monitoring, or program visualization beyond the capability of the basic environments listed. Well-developed environments provide intensive tools for debugging programs, interaction of textual/graphical representations of a parallel program, visualization support for performance monitoring, program visualization, parallel I/O, parallel graphics, etc.

The classification of a particular tool changes with time. For example, C-Linda and Fortran-Linda were developed to help C and Fortran programmers write parallel programs using the tuple spaces in Linda.

Environment Features In designing a parallel programming language, one often faces a dilemma involving *compatibility*, *expressiveness*, *ease of use*, *efficiency*, and *portability*. Parallel languages are developed either by introducing new languages such as Linda and Occam or by extending existing sequential languages such as Fortran 90, C*, and Concurrent Pascal. A new parallel programming language has the advantage of using high-level parallel concepts or constructs for parallelism instead of using imperative (algorithmic) languages which are inherently sequential.

Most parallel computer designers choose the language extension approach to solving the compatibility problem. High-level parallel constructs were added to Fortran, C, Pascal, and Lisp to make them suitable for use on parallel computers. Special optimizing compilers are needed to automatically detect parallelism and transform sequential constructs into paralled ones.

High-level parallel constructs can be implicitly embedded in the syntax or explicitly specified by users. We have identified three compiler approaches: *preprocessors*, *precompilers*, and *parallelizing compilers*, as illustrated in Fig. 11.2.

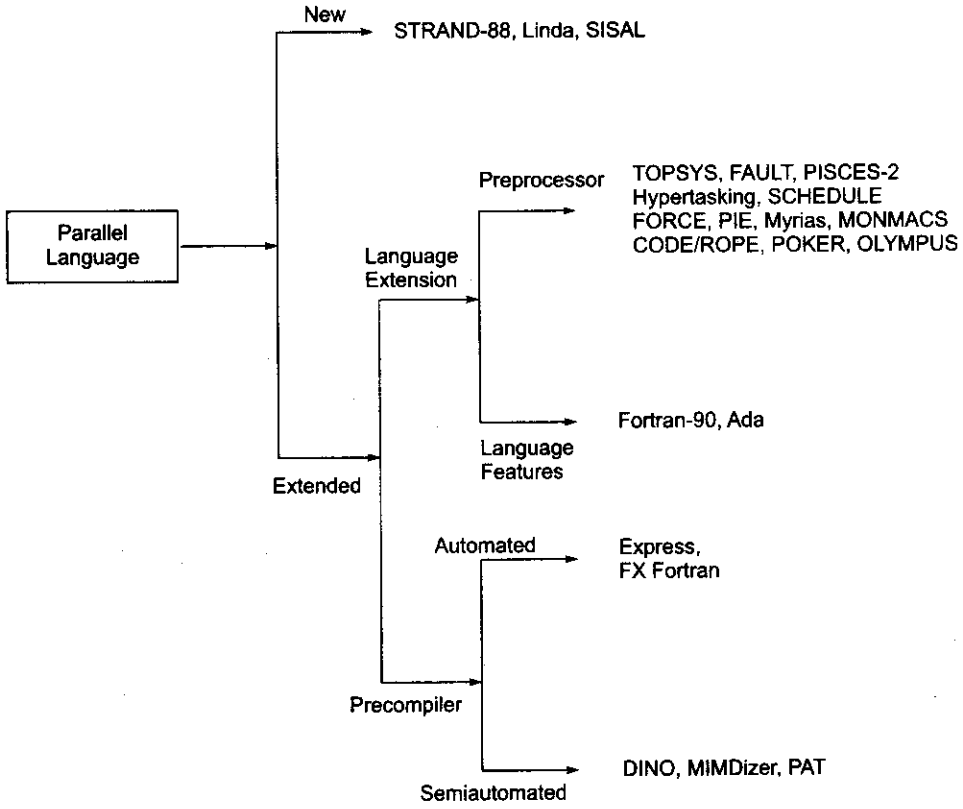


Fig. 11.2 Parallel languages and compiler technology for parallel programming at different automation levels

Preprocessors, such as FORCE and MONMACS, use compiler directives or macroprocessors. Precompilers include automated Alliant FX Fortran compilers, the Express C automatic parallelizer, and semiautomated compilers such as DINO, PAT, and MIMDizer.

Some early parallel language approaches are differentiated in Fig. 11.2 based on the degree of compilation support developed. The categorization is by no means fixed. A language tool can be upgraded from the semiautomated category to the automated category if the compiler is upgraded. Several parallel programming tools are summarized in Table 11.1.

In addition to language/compiler development, a parallel programming environment must have supporting tools to facilitate the development and testing of parallel programs. Such an environment should provide software tool sets which can be applied to different phases of the program development cycle such as for editing, debugging, performance monitoring, and tuning.

Table 11.1 *Representative Parallel Programming Tools*

Tool Name, Computing Model	*Language, OS, GUI and Environment Features*	*Hardware Platform*	*Applications Remarks and References*
MIMDizer shared-memory message-passing systems	Fortran, UNIX, VMS, X-Window, SunView, Do-loop parallelization compiler directives, array decomposition code restructuring, message-passing support.	X-MP, Y-MP, Cray 2, iPSC, Sun, IRIS, NEC.	Scientific numerical computations; Pacific Sierra Research Corp. [Harrison90] and [PSR90].
Express message-passing systems	Fortran 77, Fortran 90, C, C++, UNIX, DOS, X-Window, SunView, code parallelization, domain decomposition, communication monitoring, load balancing.	iPSC, Y-MP, Sun, nCUBE, PC, Macintosh.	Scientific (finite-element, etc.); Parasoft Corp. [Parasoft90].
C-Linda tuple-space protocol	C (Fortran called from C), UNIX, X-Window, new language extension to parallel programming.	iPSC/860, Y-MP, Sun, IBM R6000, Encore, Sequent, Apollo.	Scientific and distributed database processing; Scientific Computing Ass., Inc. [Ahuja86].
SCHEDULE shared-memory systems	Fortran, C, UNIX, X-Window, SunView, parallel algorithm development and functional parallelization.	Cray 2, Encore, Sequent, Alliant.	Scientific, numeric; [Dongarra86], University of Tennessee.

GUI = Graphics user interface.

Advances in visualization allow the programmer to visualize parallel computations through dynamic graphical animation of control flow and data flow patterns. Program visualization permits users to identify performance bottlenecks in a parallel program more easily than with a purely text-based interface.

Summary The important environment features are summarized below:

(1) Control-flow graph generation.
(2) Integrated textual/graphical map.
(3) Parallel debugger at source code level.

(4) Performance monitoring by either software or hardware means.
(5) Performance prediction model.
(6) Program visualizer for displaying program structures and data flow patterns.
(7) Parallel input/output for fast data movement.
(8) Visualization support for program development and guidance for parallel computations.
(9) OS support for parallelism in front-end or back-end environments.
(10) Communication support in a network environment.

11.1.2 Y-MP, Paragon and CM-5 Environments

The software and programming environments of the Cray Y-MP, Intel Paragon XP/S, and Connection Machine CM-5 are examined below. Readers can also refer to Chapter 13 and the plentiful material available on the web for additional information on these and other parallel computer environments.

Cray Y-MP Software The Cray Y-MP ran with the Cray operating systems UNICOS or COS. Two Fortran compilers, CFT77 and CFT, provided automatic vectorizing, as did C and Pascal compilers. Large library routines, program management utilities, debugging aids, and a Cray assembler were included in the system software. Communications and applications software were also provided by Cray, by a third party, or from the public domain.

UNICOS for Y-MP was written in C and was a time-sharing OS extension of UNIX. UNICOS supported optimizing, vectorizing, concurrentizing Fortran compilers and optimizing and vectorizing Pascal and C compilers. Besides interactive mode, it supported the Network Queueing System (NQS) for batch processing.

COS was a multiprogramming, multiprocessing, and multitasking OS. It offered a batch environment to the user and supported interactive jobs and data transfers through the front-end system. COS programs could run with a maximum of four processors up to 16M words of memory on the Y-MP system. Migration tools were available to ease conversion from COS to UNICOS.

We will describe three multiprocessing/multitasking methods—macrotasking, microtasking, and autotasking—in Section 11.2.3. The Cray Y-MP implemented all three methods and they could work together in a single program.

CFT77 was a multipass, optimizing, transportable compiler. It carried out vectorization and scalar optimization of Fortran 77 programs. The Cray assembler, CAL, enabled a user to tailor a program to the architecture of the Cray Y-MP.

Subroutine libraries contained various utilities, high-performance I/O subroutines, numerous math and scientific routines, and some special-purpose routines for communications and applications. A directory of applications software for Cray supercomputers was also made available.

Intel Paragon XP/S Software The Intel Paragon XP/S system was an extension of the Intel iPSC/860 and Delta systems. A summary of the XP/S system is given in Table 11.2. It was claimed to be a scalable, wormhole, mesh-connected multicomputer using distributed memory. The processing nodes used were 50-MHz i860 XP processors.

Paragon ran a distributed UNIX/OS based on OSF technology and was in conformance with POSIX, System V.3, and Berkeley 4.3BSD. The languages supported included C, Fortran, Ada, C++, and data-parallel Fortran.

Table 11.2 *Intel Paragon XP/S Multicomputer System*

Capacity	5–300 Gflops peak 64-bit results, 2.8-160 GIPS peak integer performance, node-to-node message routing at 200 Mbytes/s (full duplex), 1–128 Gbytes main memory, up to 500 Gbytes/s aggregate bandwidth, 6 Gbytes – 1 Tbyte internal disk storage, up to 6.4 Gbyte/s aggregate I/O bandwidth.
Node architecture	Nodes based on Intel's 50 MHz i860 XP processor, 75 Mflops, 42 VAX MIPS peak per processor, 16-128 Mbytes DRAM per node.
Operating system	Distributed UNIX based on OSF technology, conformance with POSIX, System V.3, 4.3 BSD virtual memory, simultaneous batch and interactive operation.
Programming environment	C, Fortran, Ada, C++, data-parallel Fortran, integrated tool suite with a Motif-based GUI, FORGE and CAST parallelization tools, Intel ProSolver parallel equation solvers, BLAS, NAG, SEGlib and other math libraries, interactive parallel debugger (IPD), hardware-aided performance visualization system (PVS).
Visualization	X Window system, PEX, Distributed Graphics Library (DGL) client support, AVS and Explorer interactive visualizers, connectivity to HIPPI frame buffers.

Source: Intel Corporation, Supercomputer Systems Division, Beaverton, Oregon, 1991.

The integrated tool suite included FORGE and CAST parallelization tools, Intel ProSolver parallel equation solvers, and BLAS, NAG, SEGlib, and other math libraries. The programming environment provided an interactive parallel debugger (IPD) with a hardware-aided performance visualization system (PVS).

CM-5 Software The software support and programming environment for the CM-5 are introduced here. The CM-5 designers aimed at independent scalability of processing, communication, and I/O. This aim must necessarily be supported by extensive software, languages, and application libraries.

The software layers of the Connection Machine systems are shown in Fig. 11.3. The operating system used was CMOST, an enhanced version of UNIX/OS which supported time-sharing and batch processing. The low-level languages and libraries matched the hardware instruction-set architectures.

CMOST provided not only standard UNIX services but also supported fast IPC capabilities and data-parallel and other parallel programming models. It also extended the UNIX I/O environment to support parallel reads and writes and managed large files on data vaults.

The Prism programming environment was an integrated Motif-based graphical environment. Users could develop, execute, debug, and analyze the performance of programs with Prism, which could operate on terminals or workstations running the X-Window system.

High-level languages supported on the CM-5 included CM Fortran, C++, and *Lisp. CM Fortran was based on standard Fortran 77 with array processing extensions of standard Fortran 90. CM Fortran also offered several extensions beyond Fortran 90, such as a FORALL statement and some additional intrinsic functions.

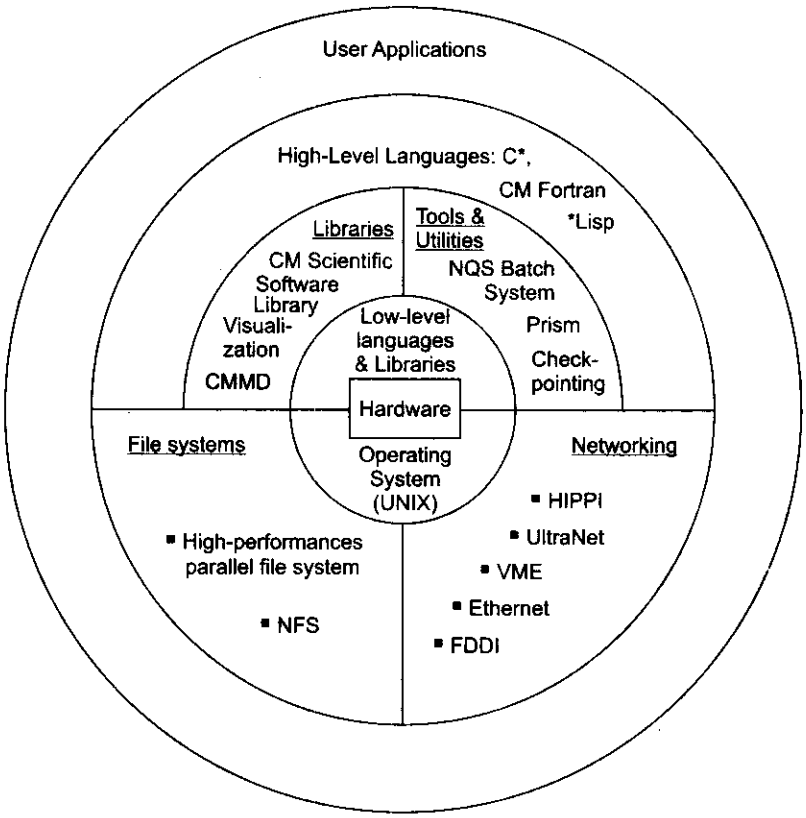


Fig. 11.3 Software layers of the Connection Machine system (Courtesy of Thinking Machines Corporation, 1992)

C* was an extension of the C programming language which supported data-parallel programming. Similarly, *Lisp was an extension of the Common Lisp programming language for data-parallel programming. Both could be used to structure parallel data and to compute, communicate, and transfer data in parallel.

The CM Scientific Software Library included linear algebra routines, fast Fourier transforms, random number generators, and some statistical analysis packages. Data visualization was aided by an integrated environment including the CMXll library. CMMD was a CM message-passing library permitting concurrent processing in which synchronization occurs only between matched sending and receiving nodes.

11.1.3 Visualization and Performance Tuning

The performance of a parallel computer can be enhanced at several stages: At the machine design stage, the architecture and OS should be optimized to yield high resource utilization and maximum system throughput. At the algorithm design/data structuring stage, the programmer should match the software with the target hardware. At the compiling stage, the source code should be optimized for concurrency, vectorization, or

scalar optimization at various granularity levels. Finally, the compiled object code should go through further fine-tuning with the help of program profilers or run-time information.

Visualization Support We consider below performance tuning at the programming, compiling, and execution stages. To probe further, we add a few performance measures to reflect the resource utilization rate for a wide class of application programs. These measures are useful in predicting the performance and in performing program tuning in an interactive manner. Special program trace methods are needed for event monitoring in the performance tuning process. A special *graphics user interface* is required to support performance monitoring, prediction, and visualization.

In the I/O and program development areas, visualization is also needed. Performance tuning requires extensive software experiments with the help of GUI and utilization support. Tuning an operating system and an application program requires effort from both ends. System tuning involves the tuning of virtual memory and process priorities, such as adjusting the resident set size and scheduling policy.

Even the best computer architect cannot guarantee the performance of a system until the machine is tested by actually running real software programs. During the architecture design stage, simulation may be used to predict performance. However, simulation experiments are often biased or restricted by theoretical load characteristics.

During the compiling and execution stages, user programs can be modified either through programmer guidance or through the use of an intelligent compiler for automatic code transformations toward optimization or vectorization. Program modification can be extended all the way back to the algorithm design stage. Choosing better data structures can often make a big difference.

Performance Tuning If special hardware/software mechanisms are available, one can also collect run-time information, such as processor and memory utilization patterns, to guide the code transformation. One can also conduct a critical-path analysis of programs in order to reveal the bottleneck. Bottleneck removal or shortening the critical path through grain packing or other techniques can improve the system performance.

Besides measuring the MIPS, Mflops, or TPS rate, performance tuning may require a check of the CPU utilization rate, cache hit ratio, page fault rate, load index, synchronization frequency, memory-access pattern, OS/compiler overhead, and interprocessor communication or data movement delays. These measures reflect the degree of matching between software and hardware.

To tune a computer system for a given application, the gap between hardware and software must be closed. *Compiler directives* can be inserted to guide code optimization. Program profilers can be used to modify the object code in multiple passes through the compiler. Program tuning at run time must be assisted by program traces and event monitoring. These may require special hardware/software support. The purpose of these traces is to produce system control parameters for more efficient processor allocation, better memory utilization, higher cache hit ratios, fewer page faults, more efficient synchronization, and lower communication overhead.

Run-time program tuning is much more difficult to implement than compile-time tuning. However, the latter lacks run-time conditions, making it difficult to predict the performance accurately. Thus both compile-time and run-time techniques are needed in the performance tuning process.

11.2 SYNCHRONIZATION AND MULTIPROCESSING MODES

Principles of various synchronization mechanisms for interprocess communication are studied first. Then we describe various modes for multiprocessing with shared memory.

11.2.1 Principles of Synchronization

The performance and correctness of a parallel program execution rely heavily on efficient synchronization among concurrent computations in multiple processors. As revealed in Chapter 6, both hardware and software mechanisms are needed to synchronize various granules of parallel operations.

The source of the synchronization problem is the sharing of writable objects (data or structures) among processes. Once a writable object permanently becomes read-only, the synchronization problem vanishes at that point. Synchronization consists of implementing the order of operations in an algorithm by observing the dependences for writable data. Shared object access in an MIMD architecture requires dynamic management at run time, which is much more complex than that of an SIMD architecture using lockstep to achieve synchronization at compile time.

Low-level synchronization primitives are often implemented directly in hardware. Resources such as the CPU, bus or network, and memory units may also be involved in synchronization of parallel computations.

We examine below *atomic operations*, *wait protocols*, *fairness policies*, *access order*, and *sole-access protocols*, based on the work of Bitar (1991), for implementing efficient synchronization schemes.

Atomic Operations Two classes of shared memory access operations are (1) an individual *read* or *write* such as Register1 $:= x$ and (2) an indivisible *read-modify-write* such as $x := f(x)$ or $y := f(x)$.

From the synchronization point of view, the order of program operations is described by *read-modify-write* operations over shared writable objects called *atoms*. An operation on an atom is called an *atomic operation*.

A hard atom is one whose access races are resolved by hardware such as Test&Set, whereas a soft atom is one whose access races are resolved by software such as a shared data structure protected by a Test&Set bit.

The atomicity of objects must be explicitly implemented by the software (on soft atoms), or the software must explicitly delegate the responsibility to the hardware (for hard atoms).

The execution of operations may be out of program order as long as the execution order preserves the meaning of the code. Three kinds of program dependences are identified below:

- *Data dependences*: WAR, RAW, and WAW as defined in Chapters 2 and 5.
- *Control dependences*: Program flow control statements such as *goto* and *if-then*.
- *Side-effect dependences*: Due to exceptions, traps, I/O accesses, time out, etc.

The correct execution order, as enforced by correct synchronization, must observe the program dependences. The atomicity of operations is maintained by observing dependences. Therefore, synchronized execution order must resolve any race conditions at run time.

Wait Protocols There are two kinds of *wait protocols* when the sole-access right is denied due to conflicts. In a *busy wait*, the process remains loaded in the processor's context registers and is allowed to continually retry. While it does consume processor cycles, the reason for using busy wait is that it offers a faster response when the shared object becomes available.

In a *sleep wait*, the process is removed from the processor and put in a wait queue. The process being suspended must be notified of the event it is waiting for. The system complexity increases in a multiprocessor using sleep wait as compared with those implementing busy wait.

When locks are used to synchronize processes in a multiprocessor, busy wait is used more often than sleep wait. Busy wait may offer a better performance if it entails less use of processors, memories, or network channels. If sleep-wait queues are managed using lock synchronization, it may be necessary for a process to wait for access to a sleep-wait queue.

Busy wait can be implemented with a self-service protocol by polling across the network, or with a full-service protocol by being notified across the network when the atom becomes available.

Fairness Policies Busy wait may reduce synchronization delay when the shared object becomes available. However, it wastes processor cycles by continually checking the object state and also may cause hot spots in memory access.

In sleep wait, the resources are better utilized, but a longer synchronization delay may result. For all suspended processes waiting in a queue, a *fairness policy* must be used to revive one of the waiting processes. Three fairness policies are summarized below:

- *FIFO*: The wait queue follows a first-in-first-out policy.
- *Bounded*: The number of turns a waiting process will miss is upper-bounded.
- *Livelock-free*: One waiting process will always proceed; not all will wait forever.

In general, the higher level of fairness corresponds to the more expensive implementation. Another concern is the prevention of deadlock among competing processes.

Sole-Access Protocols Conflicting atomic operations are serialized by means of *sole-access protocols*. Three synchronization methods are described below based on who updates the atom and whether sole access is granted before or after the atomic operations:

(1) Lock Synchronization In this method, the atom is updated by the requester process and sole access is granted before the atomic operation. For this reason, it is also called *pre-synchronization*. The method can be applied for shared read-only access. The lock granularity is a major issue in lock synchronization.

Most hardware-implemented locking applies to finer-grain physical units such as the memory module, cache, memory/cache block, etc. Lock mechanisms are described in Section 11.3.1.

(2) Optimistic Synchronization This method also updates the atom by the requester process. But sole access is granted after the atomic operation, as described below. It is also called *post-synchronization*. A process may secure sole access after first completing an atomic operation on a *local* version of the atom and then executing another atomic operation on the *global* version of the atom.

The second atomic operation determines if a concurrent update of the atom has been made since the first operation was begun. If a concurrent update has taken place, the global version is not updated; instead, the first atomic operation is aborted and restarted from the new global version.

The name *optimistic* is due to the fact that the method expects that there will be no concurrent access to the atom during the execution of a single atomic operation. This method was designed to eliminate the bottleneck created by a coarse-grain lock for the object of the first atomic operation.

This idea led to the concept of *optimistic concurrency* developed by Kung and Robinson (1981). Optimistic synchronization requires extra work in order to implement the global update operations, and the method also incurs a possible abortion cost.

(3) Server Synchronization This method updates the atom by the server process of the requesting process, as suggested by the name. Compared with lock synchronization and optimistic synchronization, *server synchronization* offers full service.

An atom has a unique update server. A process requesting an atomic operation on the atom sends the request to the atom's update server. The update server may be a specialized *server processor* (SP) associated with the atom's memory module.

Remote procedure calls and object-oriented or actor systems, in which shared objects are encapsulated by a process, provide examples of server synchronization. In a shared-memory multiprocessor using hard atoms, all three synchronization methods can be implemented, whereas only server synchronization is used in a message-passing multicomputer.

A soft atom occurs only under lock synchronization or optimistic synchronization. A specialized server processor must be used to implement server synchronization in a multiprocessor with a centralized shared memory.

In a message-passing system, the node processors with local memories can implement server synchronization without using additional server processors.

Synchronization Environment After learning about the principles of IPC and synchronization methods, we consider several implementation issues in developing parallel programs for multiprocessors.

Parallel program development hinges on how efficient synchronization is implemented with locks, semaphores, and monitors. We assess below various synchronization environments.

It is often desired to move the synchronization logic closer to the shared memory units in order to reduce bus traffic or network contention and CPU usage in the synchronization process. Such a synchronization environment may require the use of a server processor for coordinating the synchronization process and virtual memory management. The Cedar multiprocessor system has included this feature in globally shared memory units.

In general, synchronization controls the granularity of partitioned algorithms, affects the ease of writing correct programs, determines the fairness in selecting among competing processes, and ultimately influences the efficiency of parallel program execution. Special hardware and software support are needed to create an efficient and user-friendly environment.

Due to the dynamic run-time behavior, a poor synchronization environment may cause excessive waste in CPU cycles or network bandwidths, which otherwise could be more effectively used by other active processes. A poorly written parallel program may result in excessive synchronization that cancels all the advantages of parallelism. Therefore, an ideal synchronization environment should be jointly developed by designers and programmers.

11.2.2 Multiprocessor Execution Modes

Multiprocessor supercomputers are built for vector processing as well as for parallel processing across multiple processors. Multiprocessing modes include parallel execution from the fine-grain process level to the medium-grain task level, and to the coarse-grain program level.

In this section, we examine the programming requirements for intrinsic multiprocessing as well as for multitasking. Experiences using the Cray Y-MP are presented along with a programming example.

Multiprocessing Requirements Multiprocessing at the process level requires the use of shared memory in a tightly coupled system. Summarized below are special requirements for facilitating efficient multiprocessing:

- Fast context switching among multiple processes resident in processors.
- Multiple register sets to facilitate context switching.
- Fast memory access with conflict-free memory allocation.
- Effective synchronization mechanism among multiple processors.
- Software tools for achieving parallel processing and performance monitoring.
- System and application software for interactive users.

As machine size increases, the problems of communication overhead and effective exploitation of parallelism in a user program become more challenging. Meeting the challenges relies to a great extent on the development of system hardware and software support to free programmers from dealing with tedious program partitioning, parallel scheduling, memory consistency, and latency tolerance.

Multitasking Environments Three generations of multitasking software were developed by Cray Research, NEC, and other multiprocessor manufacturers. Multitasking exploits parallelism at several levels:

- Functional units are pipelined or chained together.
- Multiple functional units are used concurrently.
- I/O and CPU activities are overlapped.
- Multiple CPUs cooperate on a single program to achieve minimal execution time.

In a multitasking environment, the tasks and data structures of a job must be properly partitioned to allow parallel execution without conflict. However, the availability of processors, the order of execution, and the completion of tasks are functions of the run-time conditions of the machine. Therefore, multitasking is nondeterministic with respect to time.

On the other hand, tasks themselves must be deterministic with respect to results. To ensure successful multitasking, the user must precisely define and include the necessary communication and synchronization mechanisms and provide protection for shared data in critical sections.

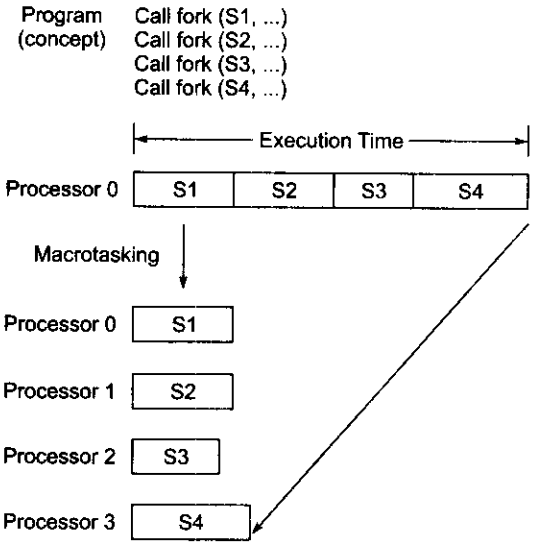
Reentrancy is a useful property allowing one copy of a program module to be used by more than one task in parallel. Nonreentrant code can be used only once during the lifetime of the program. Reentrant code, if residing in the critical section, can be used in a serial fashion and is called *serially reusable code.*

Reentrant code, which is called many times by different tasks, must be assigned with local variables and control indicators stored in independent locations each time the routine is called. The standard and familiar stack mechanism has been employed in Cray multiprocessors to support reentrancy.

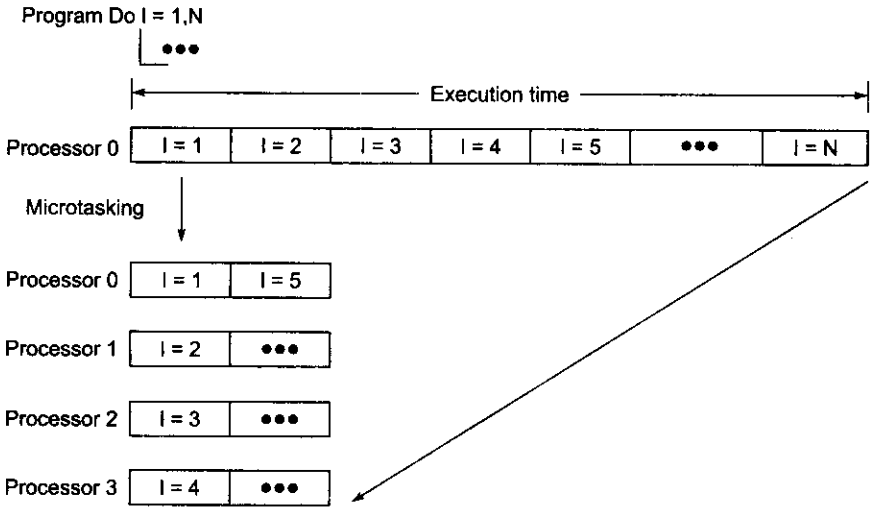
11.2.3 Multitasking on Cray Multiprocessors

Three levels of multitasking are described below for parallel execution on Cray X-MP or Y-MP multiprocessors. Multitasking tradeoffs are demonstrated by an example program execution.

Macrotasking When multitasking is conducted at the level of subroutine calls, it is called *macrotasking* with medium to coarse grains. Macrotasking has been implemented ever since the introduction of Cray X-MP systems. The concept of macrotasking is depicted in Fig. 11.4a.



(a) Macrotasking across subroutine calls



(b) Microtasking across loop iterations

Fig. 11.4 Multitasking at two different processing levels

A main program forks a subroutine S1 and then forks out three additional subroutines S2, S3 and S4. Macrotasking is best suited to programs with larger, longer-running tasks. The program interface with the

Cray Y-MP system's macrotasking capability was a set of Fortran-callable subroutines that explicitly defined and synchronized tasks at the subroutine level.

The execution of these four subroutine calls on a uniprocessor (processor 0) is done sequentially. Macrotasking can spread the calls across four processors. Note that each processor may have a different execution time. However, the overall execution time is reduced due to parallel processing.

Microtasking This corresponds to multitasking at the loop control level with finer granularity. Compiler directives are often used to declare parallel execution of independent or dependent iterations of a looping program construct.

This technique was implemented in Alliant FX multiprocessors using the **Doacross** directive. Figure 11.4b illustrates the spread of every four instructions of a Do loop to four processors simultaneously through microtasking.

When the iterations are independent of each other, microtasking is easier to implement. When dependence does exist between the iterations, the system must resolve the dependence before parallel execution can be continued. Interprocessor communication is needed to resolve the dependences.

In addition to working efficiently on parts of programs where the granularity is small, microtasking works well when the number of processors available for the job is unknown or may vary during the program's execution. Additionally, in a batch environment where processors may become available for short periods, the microtasked job can dynamically adjust to the number of available processors.

Autotasking The autotasking feature automatically divides a program into discrete tasks for parallel execution on a multiprocessor. In the past, macrotasking was achieved by direct programmer intervention. Microtasking was aided by an interactive compiler.

Autotasking demands much higher degrees of automation. Only some Cray multiprocessors, like the Cray Y-MP and C-90, were provided with autotasking software. Some compiler or assembler directives were provided to allow programmers to fine-tune their code for better performance, especially in production environments.

It should be noted that through program fine-tuning, plus the use of autotasking software support, the above multitasking feature can improve both individual job performance and overall system throughput in a production environment.

At its highest capability, autotasking should be able to achieve fully automatic multiprocessing. It should allow user programs to be automatically partitioned over multiple processors (without user intervention). Autotasking is based on the microtasking design and shares several advantages with microtasking: very low overhead synchronization cost, excellent dynamic performance independent of the number of processors available, both large and small granularity parallelism, and so on. In addition to being fully automatic, autotasking exceeds microtasking in overall performance and in the various levels of parallelism that can be employed.

Multitasking Tradeoffs Speedup from multitasking may occur only when the time saved in executing parallel tasks outweighs the overhead penalty. The overhead is very sensitive to task granularity; it includes the initiation, management, and interaction of tasks. These are often accomplished by adding code to the original code, as exemplified below.

Example 11.1 Macrotasking on a Cray X-MP dual-processor system (Courtesy of Cray Research, 1987)

This program can benefit from multitasking depending on the execution time of the subroutine SUB(J) and the overhead introduced in service routines. Before one attempts to convert serial code into multitasked code, the expected performance should be predicted to ensure a net gain.

Consider the sequential execution of the following code on an X-MP uniprocessor:

```
      Program Main
        ⋮
      Do 100 I = 1, 50
            ⋮
         Do 10 J = 1, 2
            CALL SUB(J)
10       Continue
            ⋮
100   Continue
        ⋮
      STOP
      END
```

The 100 loop has dependent iterations which cannot be executed in parallel. The 10 loop has independent iterations which are being attempted for multitasking. The following multitasked code is written for a dual-processor X-MP.

```
      Program Main
      Common/MT/IST, IDN, JOB
      CALL TSKSTART (IDTASK, T)
      JOB = 1
      Do 100 I = 1, 50
         CALL EVPOST(IST)
         CALL SUB(1)
         CALL EVWAIT(IDN)
         CALL EVCLEAR(IDN)
100   Continue
      JOB = 2
      CALL EVPOST(IST)
      CALL TSKWAIT(IDTASK)
      STOP
      END

      SUBROUTINE T
      Common/MT/IST, IDN, JOB
101   CALL EVWAIT(IST)
      CALL EVCLEAR(IST)
      IF (JOB .NE. 1) GOTO 102
      CALL SUB(2)
      CALL EVPOST(IDN)
      GOTO 101
102   RETURN
      END
```

The execution of the sequential program on one CPU consists of two parts:

Time(1 CPU) = time(Seq) + time(SUB) = (0.04 + 0.96) × (20.83 s) = 20.83 s where time(SUB) accounts for the 96% of time spent in subroutine SUB, and time(Seq) for 4% spent on the remaining portion of the program. The total run time measured on one CPU was 20.83 s.

To execute the multitasked program on two CPUs requires

$$\text{Time (2 CPUs)} = \text{time (Seq)} + \frac{1}{2}\ \text{time(SUB)} + \text{overhead}$$

The subroutine SUB was equally divided between two CPUs. This reduces time(SUB) by one-half. The overhead is estimated below with some approximation of the delays caused by workload imbalance and memory contention. The service routines TSKSTART, TSKWAIT, EVPOST, EVCLEAR, and EVWAIT were used in the Cray X-MP to establish the multitasking structure.

$$\begin{aligned}\text{Overhead} &= \text{time(TSKSTART)} + \text{time(TSKWAIT)} + 51 \times \text{time(EVPOST)} \\ &\quad + (\text{workload imbalance delay}) + (\text{memory contention delay}) \\ &= 1500\ \text{CP} + 1500\ \text{CP} + 51 \times 1500\ \text{CP} + 50 \times 200\ \text{CP} + \\ &\quad 50 \times 1500\ \text{CP} + (0.02 \times 50 \times 0.2\ \text{s}) = 0.216\ \text{s}\end{aligned}$$

where the CP (clock period) is equal to 9.5 ns. Therefore,

$$\text{Time (2 CPUs)} = (0.4 \times 20.83) + \frac{1}{2} \times (0.96 \times 20.83) + 0.216 = 11.05\ \text{s}.$$

We thus project the following speedup:

$$\text{Speedup} = \frac{\text{time(1 CPU)}}{\text{time(2 CPUs)}} = \frac{20.83}{11.02} = 1.88$$

This speedup helps decide whether multitasking is worthwhile. The actual speedup of this program as measured by Cray programmers was 1.86. This indicates that the above prediction is indeed very close.

Factors affecting performance include task granularity, frequency of calls, balanced partitioning of work, and programming skill in the choice of multitasking mechanisms.

Multitasking offers a speedup which is upper-bounded by the number of processors in a system. Because vector processing offers a greater speedup potential over scalar processing (in the Cray X-MP, vectorization offered a speedup in the range of 10 to 20), multitasking should not be employed at the expense of vectorization.

In the case of a short vector length, scalar processing may outperform vector processing. In the case of a small task size, vector processing (or even scalar processing) may outperform multitasking.

Both scalar and vector codes may be multitasked, depending on the granularity and the overhead. For coarse-grain computations with reasonably low overhead (as in the above example), multitasking is appropriate and advantageous.

11.3 SHARED-VARIABLE PROGRAM STRUCTURES

We describe below the use of spin locks, suspend locks, binary and counting semaphores, and monitors for shared-variable programming. These mechanisms can be used to implement

various synchronization methods among concurrent processes. Shared-variable constructs are also used in OS kernel development for protected access to certain kernel data structures.

11.3.1 Locks for Protected Access

Lock and unlock mechanisms are described below using shared variables among multiple processes. *Binary locks* are used globally among multiple processes. *Spin locks* are based on a time-slot concept. *Dekker's locks* are based on using distributed requests jointly with a spin lock. Special multiprocessor instructions are needed to implement these locking mechanisms.

Spin Locks The entrance and exit of a CS can be controlled by a binary *spin lock* mechanism in which the gate is protected by a single binary variable x, which is shared by all processes attempting to enter the CS.

Example 11.2 Definition of a binary spin lock

The gate variable x is initially set to 0, corresponding to the *open* status. Each process P_i is allowed to test the value of x until it becomes 0. Then it can enter the CS. The gate must be *closed* by setting $x = 1$ after entering.

```
Shared var x: (0,1)          /The spin lock/
     x := 0                  /The CS is open initially/
Process P_i for all i
     Repeat
       ⋮                     /Spinning with busy wait/
     Until x = 0
     x := 1                  /Close gate after entry/
       ⋮                     /The critical section/
     x := 0                  /Open gate after done and exit/
```

After the CS is completed, the gate is reopened. A busy-wait protocol is used in spin locks. Test and set of lock variable value must be indivisible to prevent simultaneous entries into the CS by multiple processes.

Example 11.3 Definition of a generalized spin lock with *n* possible values

One way to guarantee mutually exclusive entry is to use a *generalized spin lock* with n processes as defined below. The gate variable x is allowed to assume n integer values $(1,2, \ldots,n)$. The fact that $x = i$ implies that the gate is open only to process P_i.

```
Shared var x: (1, 2, 3, ..., n)   /The spin lock/
      x := 1                      /The CS is initially open to process P1/
Process Pi for all i = 1, 2, ..., n-1
      Repeat
        ⋮                         /Spinning with busy wait/
      Until x = i                 /Entry/
        ⋮                         /The critical section/
      x := i + 1                  /Exit/

Process Pn                        /The last process/
      Repeat
        ⋮                         /Spinning with busy wait/
      Until x = n                 /Entry/
        ⋮                         /The critical section/
      x := 1                      /Exit and return to P1/
```

Initially, the lock is open to P_1. After P_i finishes with the CS, the gate is open to P_2, and so on. The last process P_n will reset the lock to $x = 1$.

This solution guarantees mutual exclusion at the expense of longer waiting times. The processes must wait even if there are no conflicting requests at the same time.

Dekker's Protocol To guarantee mutual exclusion without unnecessary waiting, Dekker has suggested the use of separate request variables by different processes along with the use of a spin lock.

Example 11.4 Dekker's protocol for protected access to a critical section by two processes

The following program shows Dekker's solution for two processes. Each process uses a request variable (pi) to indicate if it wishes to be *inside* (1) or *outside* (0) the CS. When both processes indicate the same wish to be inside, a spin lock ($x = 1$ or 2) is used to resolve the conflict.

```
var p1, p2: (inside, outside)
Shared var x: (1, 2)
   x := 1  /The CS is initially open to process P1/
   p1 := outside; p2 := outside
Process 1                          Process 2
p1 := inside                       p2 := inside
if p2 = inside then                if p1 = inside then
begin                              begin
   if x = 2 then                      if x = 1 then
```

```
begin                                   begin
    p1 := outside                           p2 := outside
    Repeat until x = 1                      Repeat until x = 2
    p1 := inside                            p2 := inside
end                                     end
Repeat                                  Repeat
until p2 = outside                      until p1 = outside
end                                     end
  ⋮  /The critical section/               ⋮  /The critical section/
x := 2 /Open to process 2/              x := 1  /Open to process 1/
p1 := outside;                          p2 := outside
```

This scheme avoids unnecessary waiting delays whenever there is no conflicting request. The generalization of Dekker's method to a large number of processes is very cumbersome and also expensive to implement (see Problem 11.2).

Suspend Locks These types of locks use the sleep-wait protocol. A process blocked from entering a CS is removed from the processor's ready-to-run queue. Instead, the suspended process is put in a wait queue. When the suspend lock is opened, one of the suspended processes in the wait queue is reactivated using one of the fairness policies.

Spin locks or suspend locks can be implemented in a multiprocessor system using special instructions such as Test&Set, Fetch&Add, and Compare&Swap, depending on the atomic operations supported by the hardware in a given computer.

The Test&Set(x, y_i) instruction operates on two boolean variables: The spin lock x is shared by multiple processes, and the local condition variable y_i indicates the *outside* (0) and *inside* (1) wishes of process P_i. As an atomic action, this instruction sets y_i to the old value of the lock x and closes the lock (1) as follows:

Test&Set(x, y_i):
$< y_i := x; \quad x := 1 >$

If both processes simultaneously wish to enter the CS, only one can succeed in *closing* the spin lock x and thus mutual exclusion is guaranteed.

Example 11.5 Test&Set implementation of Dekker's protocol for accessing a critical section

```
Shared var x: (0, 1)
var y1, y2: (0, 1)
Process 1
    if y1 = 1 then Test&Set(x, y1)
```

```
        if y1 = 1 then Test&Set(x, y2)
            ⋮                          /The critical section/
        x := 0                         /Exit/
            ⋮                          /Noncritical section/

Process 2
    if y2 = 1 then Test&Set(x, y2)
        if y2 = 1 then Test&Set(x, y1)
            ⋮                          /The critical section/
        x := 0                         /Exit/
            ⋮                          /Noncritical section/
```

The Test&Set instruction can be used to implement Dekker's protocol, as shown in the above program in which y_i corresponds to the request variable p_i from process i for $i = 1$ and 2, respectively.

11.3.2 Semaphores and Applications

Spin locks using the busy-wait protocol may cause excessive waiting in a multiprogrammed multiprocessor system. Eliminating busy waiting in processes would make better use of the resources if a process blocked from entering a CS goes to sleep and is awakened when the CS is opened.

This improves processor utilization. Instead of using processors to execute a spinning process, they could be used more productively in executing other ready-to-run processes. *Semaphores* were developed as an improvement on the sleep-wait protocol.

Binary Semaphores Critical sections can be viewed as sections of railroad track. *Semaphores* are control signals for avoiding collisions between trains (processes) on the same track section (CS). Dijkstra (1968) introduced the use of binary semaphores for the management of concurrent processes seeking to access CSs.

A *binary semaphore* s is a boolean variable taking the value of 0 or 1. Each shared resource or CS can be associated with a dedicated semaphore. Only two atomic operations (primitives), P and V, are used to access the CS represented by the semaphore apart from initialization by setting $s = 1$.

- The $P(s)$ operation causes the value of the semaphore s to be decreased by 1 if s is not already 0; process is granted access to resource or CS. Otherwise, process enters wait state.
- The $V(s)$ operation causes the value of the semaphore s to be increased by 1 if s is not already 1; process releases the resource or exits CS.

The physical meaning of $s = 1$ is the availability of a single copy of the resource represented by s. On the other hand, $s = 0$ means the resource is being occupied. When the resource is a CS, a binary semaphore corresponds essentially to a gate or lock variable used in the sleep-wait protocol.

Counting Semaphores A *counting semaphore* s is a nonnegative integer taking $(n + 1)$ possible values 0, 1, 2, ..., n for some integer $n \geq 1$. Therefore a binary semaphore corresponds to the special case of $n = 1$. A counting semaphore with a maximum value of n acts as a collection of n permits corresponding to the use of n copies of a shared resource.

A shared resource can be a program segment, a data table, or any passive device such as memory or I/O resources. A permit is issued upon each request until all copies are taken. Formally, we define $P(s)$ and $V(s)$ on counting semaphores as follows:

- $P(s)$: If $s > 0$, then $s := s - 1$; else suspend the execution of the current process and place it in a wait queue.
- $V(s)$: $s := s + 1$. If the wait queue is not empty, then wake up one of the processes.

Intuitively, the operation $P(s)$ corresponds to the submission of a request for a permit. $V(s)$ corresponds to the return of a permit after a process finishes using a copy of the resource. Note that both operators are atomic. If the maximum value of a counting semaphore ($s = n$) has already been reached, the $V(s)$ operation will not increase the value of s beyond the upper bound n.

System Deadlock *System deadlock* refers to the situation in a multiprocessor when concurrent processes are holding resources and preventing each other from completing their execution.

In general, a deadlock can be prevented if one or more of the following four necessary conditions are removed:

(1) *Mutual exclusion*—Each process has exclusive control of its allocated resources.
(2) *Non-preemption*—A process cannot release its allocated resources until completion.
(3) *Hold and wait*—Processes can hold resources while waiting for additional resources.
(4) *Circular wait*—Multiple processes wait for each other's resources in a circular dependence situation.

Shared-Resource Allocation *P-V* operators and semaphores can be used to allocate and deallocate shared resources in a multiprocessor to avoid deadlock among concurrent processes.

Example 11.6 Resource allocation using P-V operators to prevent deadlock

Four processes are sharing six resources in the process declaration shown in Fig. 11.5a. The six resource types are represented by binary semaphores S_i for $i = 1, 2, \ldots, 6$. Allocation of the resource S_i to process P_j is requested by $P(S_i)$, and the release of resource S_i is requested by $V(S_i)$. The resource request-release pattern of each process P_j for $j = 1, 2, 3, 4$ is shown by a column of such *P-V* pairs.

All four processes can submit their requests asynchronously. It is the request ordering that leads to deadlock. The release ordering does not make any difference.

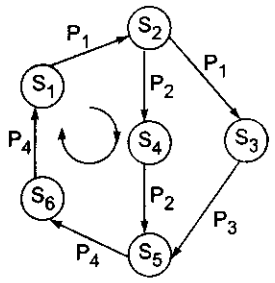
The *resource allocation graph* shown in Fig. 11.5b is a directed graph where the nodes correspond to the six resource types. An edge with the label P_k from node S_i to node S_j means process P_k is requesting resource S_j while holding resource S_i.

A cycle, $S_1 \rightarrow S_2 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_1$, in Fig. 11.5b implies the possibility of a *circular wait* among processes P_1, P_2, and P_4. To break the deadlock possibility, one can modify the resource request pattern in process 4 as shown in the fifth column.

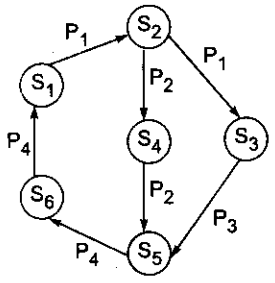
A new allocation graph results in Fig. 11.5c, where no cycle exists after reversing the edge between S_1 and S_6. Of course, this reversing should not invalidate the demand by process 4.

Process 1	Process 2	Process 3	Process 4	Process 4 (Modified)
•	P(S2)	•	•	•
P(S1)	•	P(S3)	•	P(S1)
•	P(S4)	•	P(S5)	•
P(S2)	•	P(S5)	•	•
•	P(S5)	•	P(S6)	P(S6)
P(S3)	•	•	•	•
•	•	•	P(S1)	P(S5)
•	V(S5)	•	•	•
•	•	V(S5)	•	•
•	•	•	V(S6)	V(S6)
V(S1)	V(S4)	•	V(S1)	V(S5)
•	•	V(S3)	V(S5)	V(S1)
V(S2)	•	•	•	•
V(S3)	V(S2)	•	•	•
•	•	•	•	•

(a) Four concurrent processes



(b) Resource allocation graph with circular wait.



(b) Modified resource allocation graph without circular wait.

Fig. 11.5 Shared resource allocation using P-V operators to prevent system deadlock (Reprinted from Hwang, *Proc. IEEE*, 1987)

Deadlock Avoidance *Static prevention* as outlined above may result in poor resource utilization. *Dynamic deadlock avoidance* depends on the run-time conditions, which may introduce a larger overhead in detecting the potential existence of a deadlock.

Although dynamic detection may lead to better resource utilization, the tradeoff in detection and recovery costs must be considered before choosing between static and dynamic methods.

Most parallel computers have a static prevention method due to its simplicity of implementation. Sophisticated dynamic avoidance or a recovery scheme for the deadlock problem requires one to minimize the incurred costs to justify the net gains. Breaking a deadlock problem by aborting some noncritical processes should result in a minimum recovery cost.

A meaningful analysis of the recovery costs associated with various options is difficult. This is the main reason why sophisticated deadlock recovery mechanisms have not been built into most multiprocessors.

The P and V operators are usually implemented by the underlying operating system kernel. They can also be implemented by special hardware or by software traps. Spin locks and P-V operators are both low-level, fine-grain, atomic operations which are more often used in system programming than in user programming. Only binary semaphores are used in controlling a CS. Counting semaphores are used in the deadlock-free allocation of shared resources with multiple copies each.

The main problem is that such low-level operations are error-prone and not appealing to ordinary programmers who enjoy the simplicity of using high-level language.

11.3.3 Monitors and Applications

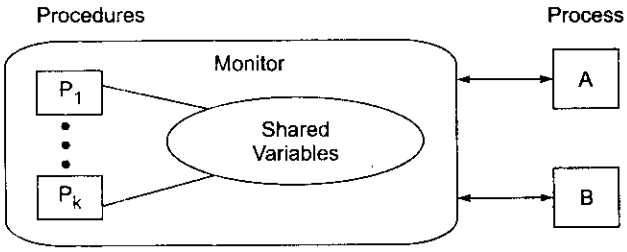
In using locks or semaphores to define critical sections, shared variables are global to all processes. CSs are embodied within processes, which may be scattered throughout the entire program. This may pose some problems in program modularization or debugging, which in turn limits parallelism.

Hoare (1974) proposed a *monitor* construct for structuring an operating system. We describe below the structure of monitors for parallel programming applications.

Monitor Structure A *monitor* is a high-level program construct for structured programming that emphasizes modularity and encapsulation. As shown in Fig. 11.6, a monitor collects shared variables and associated procedures into a single construct which allows only one process access at a time.

A typical monitor consists of the following three parts: The first part defines the monitor name and declares all local variables to be used. The second part is a collection of procedures using the variables declared. The third part is for the initialization of all local variables.

In a monitor, the CSs are removed from the bodies of user programs and become procedures or functions defined over variables confined within the boundary of the monitor. A process invokes the appropriate procedure within the monitor when it wishes to enter the desired CS. Instead of providing individual process management, a monitor behaves like a secretary in an office who provides services to a number of persons (processes). Each program can declare and use as many monitors as required.




```
Construct:
Monitor secretary (name)
    (declaration of local data)
    Procedure name (parameter list)
        begin
            ⋮        (body)
        end
    (declaration of other local procedure)
            ⋮
            ⋮
    begin
        (initialization of local data)
    end
```

Fig. 11.6 The structure of a monitor for structured programming

Producer-Consumer Implementation The monitor procedures should not access any nonlocal variables outside the monitor. An example monitor for implementing the *procedure-consumer* problem is shown below:

Example 11.7 Monitor for a producer process and a consumer process

In this monitor, two procedures are defined over the declared local variables. The producer process sends messages that are received by the consumer process. The communication between producer and consumer is handled by the *deposit* and *fetch* procedures defined below:

```
Monitor Producer-Consumer
    Buffer[0:n-1]: integer
    Inpointer, Outpointer: integer
    Not full , Notempty: Buffer conditions
    Count: integer index
```

```
Procedure Deposit(I)
    begin
        if Count = n then wait(Notfull)
        Buffer(Inpointer) := I
        Inpointer := (Inpointer + 1) mod n
        Count := Count + 1
        signal (Notempty)
    end

Procedure Fetch(I)
    begin
        if Count = 0 then wait(Notempty)
        I := Buffer(Outpointer)
        Outpointer := (Outpointer + 1) mod n
        Count := Count – 1
        signal (Notfull)
    end
begin
    Inpointer := 0
    Outpointer := 0
    Count := 0
end
```

The deposit procedure appends to or inserts messages into the communication buffer. The fetch procedure takes or fetches messages for the consumer. The producer process calls the deposit procedure for service, and the consumer process calls the fetch procedure for service.

Only one process (either producer or consumer) can call the desired procedure at a time. A monitor is not itself a process. A monitor is a static module of data and procedure declarations. The producer and consumer are active processes which must be programmed separately as shown below:

```
Producer                Consumer
    I: integer              I: integer
    repeat                  repeat
    begin                   begin
        Produce(I)              Fetch(I)
        Deposit(I)              Consume(I)
    end                     end
```

Monitor Applications All the operations of semaphores can be simulated by monitors, and vice versa. Monitors have been suggested for use in a number of parallel programming applications.

Brinch-Hansen (1977) used monitors for Concurrent Pascal programming. In general, a parallel program may contain two different kinds of program modules: *active processes* and *passive monitors*. As in the producer-consumer example, all shared variables are defined within the monitors; interprocess communications are handled by calling procedures in the same monitor. Different groups of processes may use different monitors (secretaries) for different types of services (procedures) which may require special shared resources (local variables).

Several distinct advantages are observed in using monitors to support IPC or interprocessor synchronization. First, all dedicated services (procedures) are placed together within each monitor. Therefore, a calling process can be freed from worrying about these procedure details. Second, the monitor designer does not have to worry about how many user processes may access it. Third, monitors provide modularity in program debugging and maintenance.

For example, one can program a disk scheduler as a dedicated monitor. All user processes can submit requests to the disk scheduler, and only one is serviced by a driver procedure at a time. Monitors hide information within their boundaries, which implies the potential for concurrent object-oriented programming and efficient handling of abstract data types. Programs using monitors are not only easier to debug but also are easier to use in exploiting parallelism. The localization of shared variables is an important asset in concurrent programming.

Monitors can be implemented directly with data and procedure declarations, or indirectly through the use of semaphores such as using *P-V* primitives to declare the entry and exit of a monitor. Monitors can also be implemented with the help of a kernel of data structures and routines. A monitor kernel may include special primitives for *entry*, *wait*, and *signal* operations. Wirth (1977) proposed a Modula kernel for implementing monitors when using the language for parallel programming.

11.4 MESSAGE-PASSING PROGRAM DEVELOPMENT

Multicomputer programming demands the distribution of computational load and data structures to various node processors for balanced parallel processing. Message-passing paradigms are needed for internode communications. Over the last two decades, message-passing has gained importance as a means of achieving distributed computing.

Three program/data decomposition techniques, namely, *domain*, *control*, and *object decomposition*, are presented for programming a multicomputer. In each case, example problems are given to illustrate the decomposition technique involved.

11.4.1 Distributing the Computation

The key to using a multicomputer system is to distribute the computations among an ensemble of computer (processor-memory) nodes. Each node executes its own program, and all nodes are interconnected by a network. Concurrent processes created at different nodes communicate by passing messages. In this section, we assess the basic programming environment for multicomputers. We study program tuning to achieve load balancing directed toward higher performance.

Host and Node Environments The programming environment of a multicomputer may include an optional host run-time system and resident operating systems in all node computers.

For example, a Cosmic Environment was developed for the hypercube computer at Caltech. The host environment was a UNIX processor and used UNIX and language processor utilities to communicate with the node processes and other host processes through messages.

A separate OS kernel was located in each node computer that supported multiprogramming, with an address space confined to local memory. Many node processes could be created at each node. In fact, the total number of node processes could be greater than the number of node computers in the system.

All node processes executed concurrently in different physical nodes or interleaved through multiprogramming within the same node. Node processes communicated with each other by sending or receiving messages.

The *Cosmic kernel* (later modified to the Reactive Kernel) at Caltech was one such node operating system that supported this process-model programming.

There were no shared variables between node processes in the Cosmic kernel, even if they resided in the same node. The node processes did not have access to input/output devices, and all I/O activities were handled by the hosts.

The Cosmic Cube programming environment did not use new programming languages. Existing sequential programming languages such as C, Pascal, Fortran, Lisp, Assembly, etc. were used to write process codes. A library of C functions and procedures was developed to control message-passing and process-spawning operations.

This approach used explicit parallelism built on top of existing compiler technology. Many interesting Cosmic features were built into or modified in commercial systems such as the Intel iPSC and nCUBE computers. Various multicomputer programming environments differ mainly in the languages and message-passing paradigms used.

In programming a multicomputer, the process involves separation of the user interface from the computational kernel, leaving the user interface on the host or a designated node, moving the kernel to each node, and adding a message interface between them.

In order to distribute the computation, the programmer chooses a decomposition method and then maps the decomposition to all the nodes. A node-to-node communication protocol must be established. Finally, one needs to balance the load and reduce the communication/computation ratio.

Message Types and Parameters Example 10.1 demonstrated the need for message passing. In a multicomputer program, a process must distinguish between a number of different message types. A particular field of all messages can be reserved to carry a *message type* (identified by an integer).

Different message types may demand different actions by the sending or receiving processes. Messages of different types are handled in a specific order. For example, the Cosmic Environment/Reactive Kernel dispatched messages according to the types received, and supported customized message functions on top of the X-window primitives.

Let us examine the basic message parameters associated with *send* and *receive* system calls. The message-passing primitives are specified by:

send (*type*, *buffer*, *length*, *node*, *process*)
receive (*type*, *buffer*, *length*)

where *type* identifies the message type, *buffer* indicates the location of the message, *length* specifies the length of the message (in bytes), *node* designates the destination node, and *process* is the process ID at the destination node.

The *send* and *receive* primitives are used by the sending and receiving processes, respectively. Therefore, the buffer field in *send* specifies the memory location of the message to be retrieved from. On the other hand, the buffer field in *receive* specifies where the arriving message will be stored.

Once stored in a local memory, a message can be retrieved only by the local processor. No remote memory access is allowed in a pure message-passing multicomputer. The implementation requirements of the two message-passing models are studied below.

11.4.2 Synchronous Message Passing

The message-passing process involves a sender and one or more receiver(s). When a process sends a message, the system must decide a number of issues: first, whether the receiver should cooperate or be ready to receive it; second, whether the communication path has been established or not; and third, whether one or more messages can be sent to the same destination node or to multiple destinations.

In a synchronous communication network, the sender process and receiver process must be synchronized in time and space. Time synchronization means both processes must be ready before message transmission can take place. Space synchronization demands the availability of a communication path, i.e. a sequence of connected channels from source to destination.

The simplest implementation allows only one message on a communication channel at a time. No buffers are used in such a communication network. Therefore, blocking is possible if the channel requested is busy or in error. For this reason, synchronous message passing belongs to the class of blocking communication systems. The correct protocol must be adopted to ensure the coupling of the sender and receiver in time and space. Blocking may take place very often in such a network. How to minimize the delays caused by blocking is a major issue to be considered.

Synchronous message handling simply ignores blocked messages, assuming no buffers are used with the communication channels. This scheme has been implemented as one of two possible message-handling modes in the Intel iPSC systems. The idea is to halt further execution of instructions in a process until the desired message is sent or received.

When a process issues a request to receive a message, the process, being blocked, executes no further instructions until the expected message has been received. If the message arrives ahead of the request, the receiver will wait only until the message is copied into the local memory from the system buffer in which it was temporarily stored.

Similarly, when a sender initiates a message transfer, the sending process is blocked until the message is copied from local memory into the message-passing network by the node kernel. Synchronous message handling is easier to implement but may not result in the highest performance possible in a given network.

The Ada Experience Example synchronous message-passing programming systems include the unbuffered Ada system, which uses a *rendezvous* concept to synchronize the sender with the receiver. In such a system, the early arrival at the rendezvous must wait for the late arrival. Ada uses a *name-addressing* scheme in which (Node, PID) is used to identify a process residing at a node.

The Ada system allows two-way data flow during a single rendezvous. A process calls another process by name without divulging its own identity. Ada implements a *select* primitive, which allows a process (task) to conditionally select an entry call to execute among multiple entries.

Programming with the rendezvous concept can easily implement a *remote procedure call* (RPC) besides the *select* option. It also supports dynamic task creation and priorities in task selection.

The Occam Experience The CSP (communicating sequential processes) model proposed by Hoare employs a selective synchronous scheme based on a tightly synchronized form of message passing. In 1988 CSP was modified and called the Occam system by INMOS/Transputer developers, based on a *channel-addressing* scheme. This scheme established a communication path between sender and receiver by directional channel tracing. The Occam system used one-way data flow.

In a synchronous message-passing system, the number of completed *receive* operations is identical to the number of completed *send* operations at the other end. The blocking problem can be avoided or alleviated using buffers in the message-passing network.

Buffering is like a telephone system with an answering machine or like a fax machine. The sender and receiver do not have to be available at the same time, and yet they can communicate with each other.

11.4.3 Asynchronous Message Passing

This programming paradigm requires the use of buffers on communication channels or the use of a global mailbox. Message sending and receiving do not have to be synchronized in time or space. Nonblocking communication is possible if sufficiently large buffers are used along the communication channels.

Blocked messages are buffered for later transmission. This system is like a postal service system using many mailboxes as buffers. No synchronization is needed because the sender does not have to know if and when a message is received. The receiver expecting a message will not be suspended from regular execution.

Even when we consider a buffered asynchronous system nonblocking, the system may eventually be blocked due to the use of limited-size channel buffers. If the network traffic is not heavy or saturated, moderate-size buffers will lead to essentially nonblocking communications. Most advanced message-passing networks use asynchronous communications for the sake of better resource utilization and shorter communication delays. An arbitrary delay may result in a buffered communication scheme under very heavy load.

Caltech's Cosmic programming environment supported asynchronous message passing. Intel's iPSC system also supported asynchronous message handling. Asynchronous *receives* allow processes to alert the node kernel that certain messages are expected and should be delivered as soon as they arrive.

In the meantime, the receiving process can continue its work if needed. An asynchronous *send* allow a sender to alert the kernel that it wants to send a message, but the process does not wait until the message is sent. Obviously, asynchronous communication is more efficient and sometimes faster.

The Linda Experience The *Linda* programming system also operates asynchronously. Uncoupling of the sender and receiver in time and space is expected. Linda is based on a global mailbox or bulletin-board concept for achieving asynchronous communications. This is done through the use of a *tuple space*, which is logically shared by all concurrent processes.

The tuple space consists of *tuples*, which are typed data sequences. Any process can add a tuple to the shared tuple space or remove a matching tuple from the space. If no match is found, the process suspends until a match is found later. It can also read a copy of a matching tuple without removing it from the space.

In many ways, the tuple space behaves like a bulletin board where anybody can add a notice (tuple), read a notice, or remove a notice. The tuple space, therefore, becomes a global mailbox which can be accessed by all processes for the purpose of either sending, broadcasting, or receiving messages (tuples).

Concurrent programming works in Linda by pattern matching on the tuple signatures (tag fields). The concept of tuple space can be implemented on either shared-memory multiprocessors or distributed-memory multicomputers.

Interrupt and Lost Messages Interrupt messages are a special form of asynchronous message handling. Instead of continuing working while waiting for a regular message to arrive, interrupt handling is carried out immediately by the receiver without having the receiving process post a *receive* when it is ready to receive. After the interrupt is serviced, the interrupted process may resume its original work.

Messages are often lost in a message-passing system. When an incoming message is not expected or needed, it may be ignored and thus lost. Messages directed to the wrong process or wrong node will not be found or retrieved by the intended process, and the node kernel may not be able to cope with the problem.

These messages may end up in the system message buffers and eventually be lost. This may not be the fault of a programmer. Special detection aid or debugging tools are needed to inspect message buffers and to correct possible system configuration or programming errors.

11.5 MAPPING PROGRAMS ONTO MULTICOMPUTERS

This section describes program decomposition techniques based on data domains, control structures, functionality, and object-oriented concepts. In all cases, we aim at performance tuning and enhancement of message-passing programming. At the end, we characterize the environment for heterogeneous programming.

11.5.1 Domain Decomposition Techniques

Programming a multicomputer requires three major steps: *decomposition*, *mapping*, and *tuning*. The goals are to balance the load, minimize communication overhead, reduce sequential bottlenecks, and make the program scalable. Decomposition of the π calculation is a perfect example of domain decomposition .

In general, if a calculation is based on a large, static data structure and the amount of work is about the same for each data element, then one should partition the data structure evenly.

The resulting programming technique is called *domain decomposition*. This technique can be used with a wide range of applications, including physical modeling, matrix computations, and database/knowledge base management.

Perfect Decomposition To choose the best decomposition method for a given application, one needs to understand the mathematical formulation, the data domain, the algorithm used, and the flow of control (communication pattern). Certain applications fall naturally into the *perfect decomposition* category. Such parallel applications can be divided evenly into a set of processes that require little or no communication with each other.

This category is the easiest to decompose. The π computation is a good example of a perfect decomposition in which only the partial sums need to be communicated. The order in which these summations are performed is immaterial, and thus interprocess synchronization is unnecessary.

Perfect decomposition often leads to SPMD operational mode. In other words, the same node program is replicated at all the nodes. Real applications of this kind range from the modeling of proteins with thousands of atoms and millions of possible configurations to financial speculation involving investments subject to parallel evaluation of multiple hypothetical portfolio cases simultaneously.

Perfect decomposition requires very little communication overhead, and the balanced computations often result in nearly 100% efficiency. This kind of decomposition requires the least amount of programmer effort.

Often the same sequential program running on a single processor will be running on all node processors, each with a different data set. Once an interprocess dependence relationship exists, the conditions for perfect decomposition may be compromised.

Domain Decomposition The key to domain decomposition is the *regularity* of the data structures involved. Three kinds of problem domains are identified below as natural candidates for domain decomposition:

(1) *Static data structure*—For example, matrix factorization for solving a large finite-difference problem on a system with a regular network topology.
(2) *Dynamic data structures tied to a single entity*—For example, in a many-body problem, subsets of the bodies can be distributed to different nodes. Through gravitational forces, the bodies may be interacting with each other and moving in space. The calculation for each body can stay on the original node assigned.
(3) *Fixed domain with dynamic computations within various regions of the domain*—For example, a program that models fluid vortices, where the domain stays fixed but the whirlpools move around.

Three major steps are specified below to decompose the domain of a given application:

(1) Distribute the subdomain of data to various nodes.
(2) Restrict the computation so that each node program updates its own subdomain of data.
(3) Put the communication in node programs.

The way to implement the above steps is first to port the sequential program to various nodes. The porting procedure involves the following operations: Compile and test the existing sequential program on one node and then run multiple copies of the same program on many nodes at once, after putting in the communication commands required. Localize the node program execution over its own data. Finally, tune the program to enhance the performance.

Example 11.8 LINPACK matrix factorization using domain decomposition (Justin Rattner, Intel Scientific Computers, 1990)

Gaussian elimination is often performed in factoring a square matrix in the LINPACK linear equations package. The regular nature of this algorithm and the regularity of the domain make it inherently parallel and suitable for domain decomposition.

Distributing the domain only requires partitioning the characteristic matrix into sections and distributing them among the processor nodes. We will first examine the sequential LINPACK Gaussian elimination

algorithm. After determining the matrix distribution, we then distribute the computation, resulting in a parallel matrix factorization algorithm.

The LU factorization is a triply nested loop: The outer loop controls how much of the matrix remains to be factorized. At each iteration, the remaining part of the matrix is a smaller submatrix in the lower right-hand corner. The elimination process requires that information from one node be broadcast to all the other nodes. A sequential factor algorithm in LINPACK is as follows:

Sequential Factor Algorithm
for $i = 1$ to $n - 1$ **do**
  Find max$[A(i, i)$ to $A(n, i)]$ in ith column,
  Swap rows to make $A(i, i)$ the pivot,
  Divide $A(i + 1, i)$ to $A(n, i)$ by $A(i, i)$,
  for $i = i + 1$ to n **do**
    for $k = i + 1$ to n **do**
      $A(k, j) \leftarrow A(k, j) - A(k, i) \times A(i, j)$
    end of k-loop
  end of j-loop
end of i-loop

This sequential code can be executed directly on a single node. The next step is to distribute the matrix elements. The matrix is distributed by columns among the processor nodes. The matrix domain is mapped to the nodes so that all the processors own approximately the same number of columns of the matrix.

Figure 11.7 shows a column-wrapped mapping of a 12×12 matrix onto four processors. Once a column becomes the pivot column, it requires no further computation. By using this column mapping, all processors can be kept busy during most of the computation, and a new processor owns the pivot column at each step.

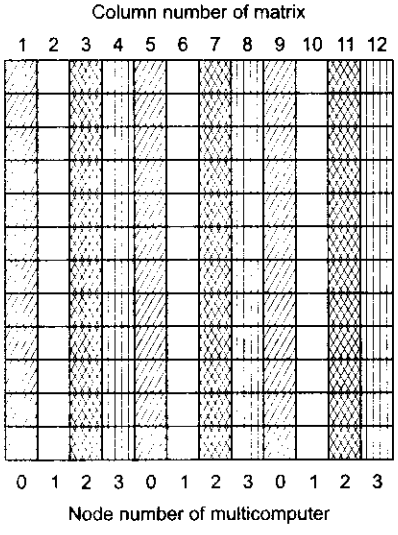


Fig. 11.7 Column-wrapped mapping of a 12×12 matrix onto four nodes of a multicomputer (Courtesy of Justin Rattner, Intel Scientific Computers, 1990)

The *parallel factor algorithm* is designed to make the pivoting processor control the computation at each outer-loop iteration. The processor owning the first column finds the pivot element, swaps the pivot row with the first row, and divides the pivot column by the pivot, just as in the sequential factor algorithm.

After the first iteration, the pivot processor broadcasts the pivot column and the pivot row number as a message to all the remaining processors. Any processor receiving this message swaps its pivot row with its first row. Then all processors perform the remaining subtraction and scaling from each remaining column simultaneously.

This parallel factorization algorithm is specified below, where n is the order of the matrix, N is the total number of nodes, i is the global index, h is a local index with an initial value of 1, and p is the index of the processor. The value of p is different for each processor and is in the range $0 \le p < N$. In addition, all processors use the same node program as stated below.

Parallel Factor Algorithm
$p \leftarrow$ index of this processor
for $i = 1$ to $n - 1$ **do**
  if $(i - 1)$ mod $N = p$ **then**
    Find max in the hth column
    Swap row i and h to make $A(i, i)$ the pivot
    Divide $A(i + 1, i)$ to $A(n, i)$ by $A(i, i)$
    Broadcast pivot column, $A(i + 1, i)$ to $A(n, i)$,
      and pivot row number h
    $h \leftarrow h + 1$
  else
    Receive the pivot package
    Swap rows
    Scale and **subtract** the pivot column from each remaining column
end of i-loop

The above parallel algorithm is modified from the serial algorithm by the addition of a *test* to determine the pivot processor and system calls to *send* (broadcast) the pivot information to all the processor nodes. This simple change may result in a maximum speedup proportional to the number N of processors used in the multicomputer.

Performance Tuning From a successive pivoting point of view, the above algorithm is essentially sequential. While a pivot column is being determined, all the remaining processors are waiting for it. When the pivot is finally broadcast, all the processors can subtract in parallel and then proceed with the next pivot node, etc.

From a performance point of view, the load is not fully balanced across the nodes. The sequential bottleneck is in the successive pivoting processors. The mapping of the matrix columns in a modulo fashion has already provided a better load balance than assigning adjacent columns to the same processor.

The performance can be further tuned by reordering the algorithm to speed up the pivot and broadcast process, or by dividing the row in parallel instead of the column. Furthermore, one can change the granularity

by using a *block algorithm* to obtain fewer and longer messages, which will improve the communication/computation ratio.

In other applications, such as seismic processing, finite-element analysis, vision integration, and multidimensional complex FFT, the performance bottleneck is in different areas such as extensive synchronization delays or dynamically changing data structures. In each case, performance tuning will be focused differently.

11.5.2 Control Decomposition Techniques

When the domain and data structure are irregular or unpredictable, we cannot apply domain decomposition. One alternative is to focus on distributing the flow of control of the computation rather than on distributing the domain.

An example of a domain that is not suitable for domain decomposition is the irregular search space created by a game tree where the branching factor varies from node to node. The search tree must be dynamically assigned to maintain a balanced load.

In general, *control decomposition* is used for symbolic processing problems such as those in artificial intelligence applications. In this section, we will study several control decomposition strategies, including *functional decomposition* and a *manager-worker approach.*

Functional Decomposition An algorithm can be visualized as a set of interconnected functional modules. The flow of control is indicated by directed edges in the diagram. For small problems, these functional modules tend to be executed sequentially, one after another. However, large problems may have significant parallelism between the modules.

Example 11.9 Image understanding with functional decomposition

Figure 11.8 shows an example of functional decomposition for image understanding.

Image enhancement, edge detection, pattern recognition, and scene and motion analysis are processed by four functional modules in a pipelined fashion (with a feedback branch, if adaptively done) over a sequence of image frames.

The functionality requires the application of different decomposition techniques, depending on the data structures and computations involved. One needs to add special message interfaces between different functional modules.

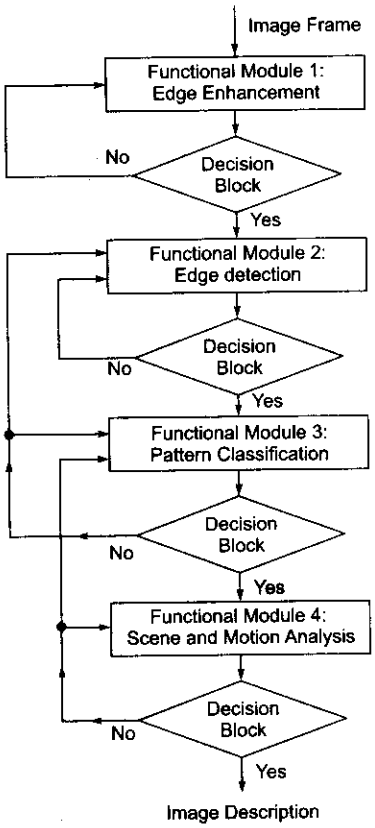


Fig. 11.8 Functional decomposition for image understanding

Usually different functional modules are assigned to different processor nodes. Some nodes will be floating-point-intensive, some for symbolic manipulation, some for input/output activities, etc.

Manager-Worker Approach This is a divide-and-conquer technique. The idea is to divide the application into tasks, not necessarily having the same size, and use one of the processes to serve as a manager node and the rest as worker nodes.

As illustrated in Fig. 11.9, the manager is responsible for dispatching tasks out as worker nodes become available. The manager must also communicate with the user or the host node for input/output operations.

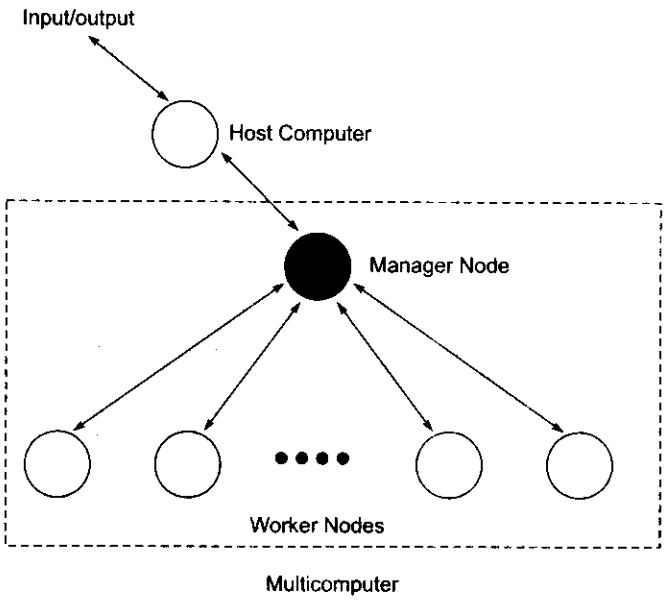


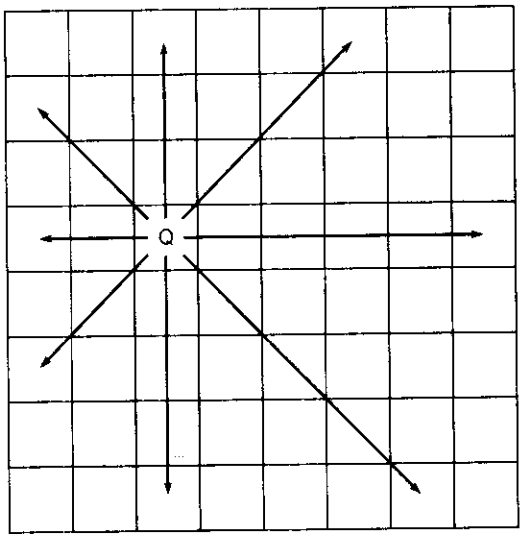
Fig. 11.9 Manager-worker mapping for control decomposition (Courtesy of Justin Rattner, Intel Scientific Computers, 1990)

The manager functions include management of global data structures, maintaining a list of subprograms (tasks), and assigning problems/tasks to workers. The worker node, once it becomes available, should request the job, receive the job, and perform the assigned task. The manager must perform dynamic load balancing among the workers in order to enhance the performance of the entire system.

Example 11.10 Solving the *N*-queens problem on a multicomputer

A typical application of the manager-worker decomposition is to solve the N-queens problem on a multicomputer. The problem is to find all possible solutions for placing N queens on an $N \times N$ chessboard so that no more than one queen is on each row, each column, or each diagonal.

In other words, each row, column, or diagonal must have exactly one queen and no queen can attack another, as illustrated in Fig. 11.10. The way the *N*-queen problem is solved is to generate a search tree using the workers to solve each leaf node of the search tree.



Problem: Find all solutions for placing N queens on an $N \times N$ chessboard so that there is exactly one queen on each row, each column, and each diagonal.

Fig. 11.10 *N*-queens problem: An example of manager-worker decomposition (Courtesy of Justin' Rattner, Intel Scientific Computers, 1990)

The manager builds and maintains the top level of the search tree, assigns workers to build different branches of the tree, and keeps track of the total number of solutions (chessboard patterns) generated. Each worker should be able to split the problem into subproblems and to solve a subproblem.

Once a branch is too heavily extended, the worker can report to the manager which will off-load subtree operations to other available nodes. Close communication between the manager and the workers is necessary to keep up a well-belaneed processing of the search tree on a dynamic basis.

Performance Tuning One technique for tuning the performance is to use double-buffer messages in a manager-worker decomposition. The idea is to send a worker two pieces of work the first time. As soon as the worker is finished working on the first piece, another piece is readily available.

Once the communication/computation issues have been well balanced, the results from the first piece of work are returned to the manager who can send over another piece of work before the worker finishes the second piece. There is always one job waiting in the worker's queue, and thus workers are kept busy all the time.

A potential problem with the manager-worker approach is that the manager may become the bottleneck. According to Intel iPSC experience, up to 50 workers managed by a single manager did not create a serious bottleneck problem as long as a good communication/computation ratio was maintained.

Clearly, for a multicomputer consisting of thousands of processors, the manager bottleneck problem can become more serious. One can then consider providing a hierarchy of managers.

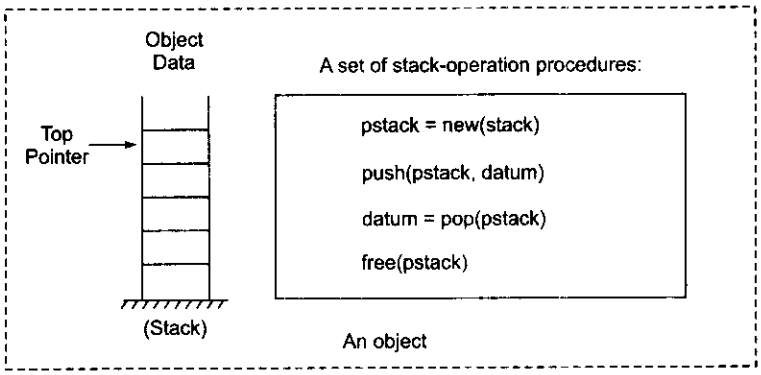
Another possible solution is to use floating managers or multitasking processors, which can execute both a worker process or a manager process on the same processor. These options must be carefully analyzed and experimented with before they can be adopted in real applications.

11.5.3 Heterogeneous Processing

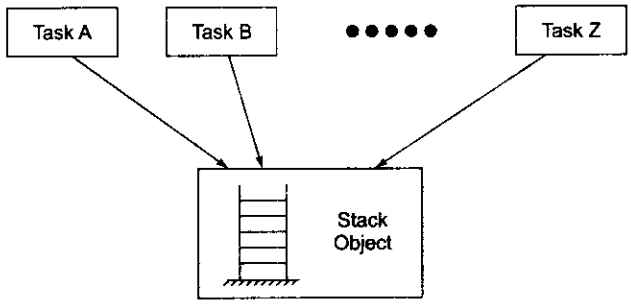
In this section, we learn how to combine object-oriented programming with message-passing techniques for distributed computing applications. We first characterize objects in relation to parallelism. Then we illustrate the *object decomposition* concept using an air traffic control simulation example.

Finally, we present the concept of *layered parallelism* using a seismic monitoring example which involves all the decomposition techniques we have learned. Such concepts may be needed to solve very large-scale problems.

Objects and Parallelism The object-oriented approach to parallel programming offers a formal basis for decomposing the data structures and threads of control in user programs. In what follows, we define objects and reveal the relationship between objects and parallel processing.



(a) Defining a stack as an object



(b) Shared access to a stack object by multiple tasks

Fig. 11.11 A stack object and its shared access by multiple tasks

The idea of objects comes from *data abstraction*, in order to hide low-level details from programmers. An object encompasses a set of logically related data and a set of procedures which operate on the object's data, as illustrated by the example in Fig. 11.11a.

The example shows that temporary storage in the form of a stack can be treated as an object consisting of a last-in-first-out queue of data which can be pushed down or popped up in its management. It should be noted that an object type (class) is conceptually different from instances of the object type.

Those instances, called objects, are the ones used in program execution. Only the object's procedures have access to the object's data. A programmer can be freed from knowing the detailed implementation of the objects.

This simplifies program debugging and testing efforts, and offers modularity in program development, which are all desired features for parallel programming.

Shared data structures can be organized as objects. Shared access to a stack object by multiple tasks is shown in Fig. 11.11b. A parallel program may be composed of multiple threads of execution that access both private and shared data.

Threads of execution can be allocated as instances of a task type. If tasks must communicate by exchanging messages, the shared object is accessed via message passing, which must be synchronized by a system-defined class of queue objects.

Object Decomposition This technique offers natural advantages for parallel computers. It avoids the use of global variables, simplifies the program/data partitioning process, and provides higher granularity of interaction among objects through the use of predefined procedures for accessing objects.

Ada and C++ both support data abstraction in object-oriented programming on message-passing systems. The following air traffic simulation system explains the key concepts behind object decomposition for parallel programming. The goal of the simulation is to measure the effects of scheduling, weather, etc., on air traffic flow control.

Example 11.11 Air traffic simulation on a multicomputer

Three fundamental object types are identified in Fig. 11.12. First, the airports contain information regarding their location, runways, etc. Second, airplanes contain information related to position, velocity, fuel capacity, etc. Third, air space sectors are pre-specified by the American Flight Agency.

Objects of these types are manipulated by four task types: The air dispatchers allocate airplanes and pilots (scheduling). The pilots operate the airplanes. The air traffic controllers manage the safe use of airports and airspace sectors to avoid collisions.

The tasks interact by invoking procedures on shared airplane, airport, and airspace objects. The communication among the tasks and between tasks and objects can be simulated by a message-passing multicomputer. Appropriate protocols must be established in these message-passing operations.

The use of multiple tasks with separate functions in this application is also an example of control decomposition. The computational complexity of this air traffic simulation problem is controlled by partitioning the airspace into sectors.

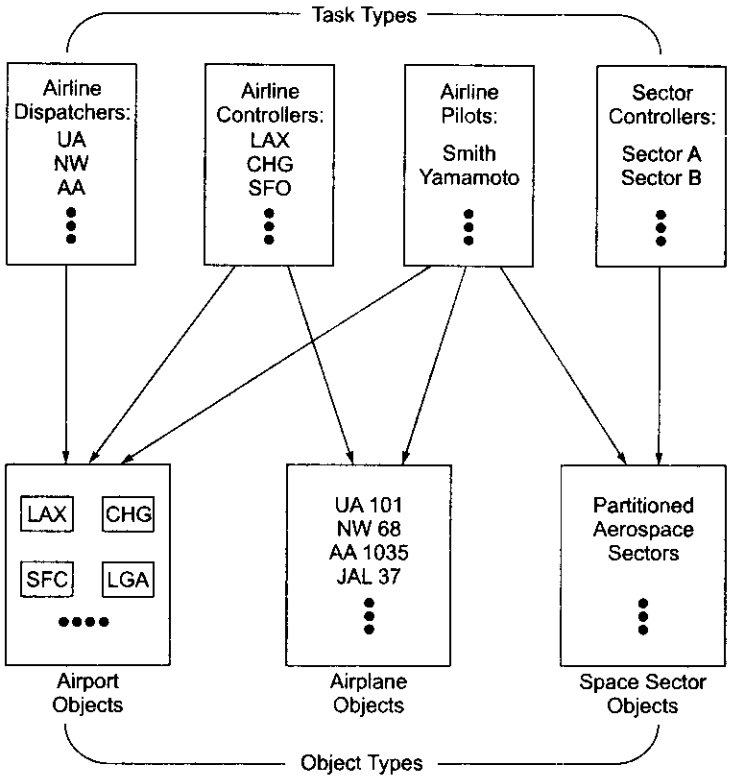


Fig. 11.12 Air traffic simulation using decomposition techniques

The air traffic controller manages the separation of airplanes within each sector. When an airplane crosses between two sectors, the controller must pass the duty to the next sector controller on the route of the flight.

Sector partitioning of the airspace is indeed a domain decomposition of the problem. Different sector controllers are simulated at different nodes of a multicomputer. In order to conduct real-time simulations, all the object information must be retrieved from the local memory without page faults from the backup store.

For example, distribution of the airports should be made to associate the airports and their controllers within the same sector node. As airplanes fly from one airport to another, the objects are sent as messages among nodes. Load balance can be achieved by defining the sector boundaries, interleaving sector objects, or allowing sector migration dynamically as traffic becomes dense.

Heterogeneous Processing A growing trend in computing is to use a network of resources for *heterogeneous processing* (HP). A large-scale complex problem can be solved by combining a number of computers of various kinds in a network environment.

On a multicomputer-based network, HP can be practiced using a combination of decomposition techniques. In what follows, we describe computations with embedded parallelism at various processing layers.

Layered Decomposition In solving large and complex problems, we may have to employ a programatic layered approach to extracting parallelism using different decomposition techniques at different levels. Such an approach can be called *layered decomposition.*

The functional approach to image understanding (Fig. 11.8) can be considered a four-layered decomposition. As machine size reaches thousands of nodes, we have to develop programming techniques on that scale.

The layered approach certainly addresses the scalability problem with foresight. Example applications include weather simulation, fluid flow, structural analysis, molecular dynamics, and seismic monitoring.

Example 11.12 Heterogeneous processing based on layered parallelism

The problem of seismic monitoring requires the processing of a large amount of seismic data using a layered approach consisting of three levels, as illustrated in Fig. 11.13.

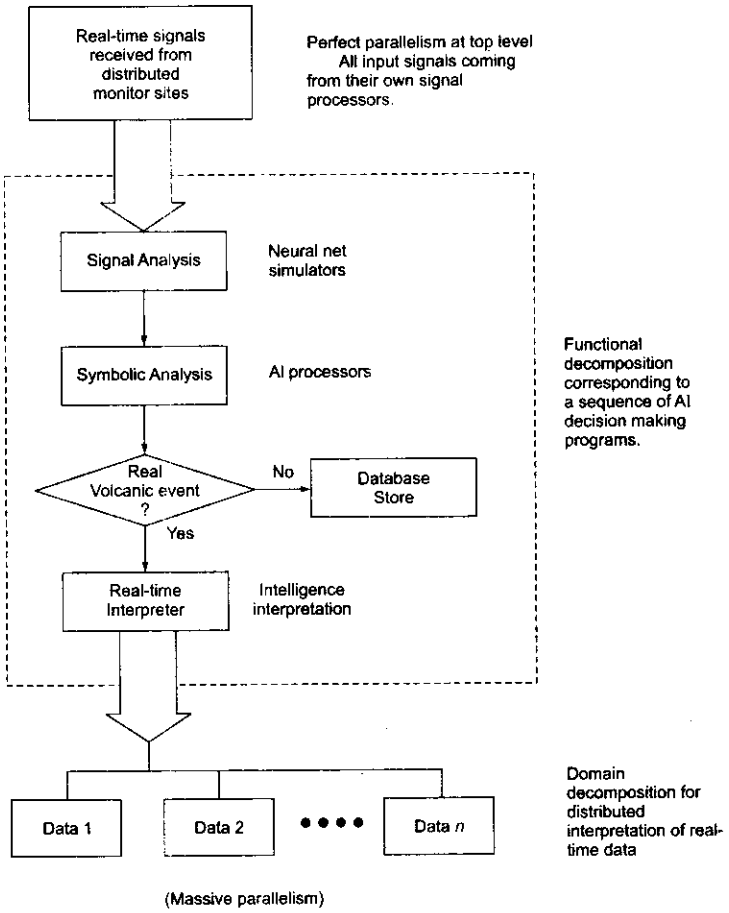


Fig. 11.13 Layered parallelism in decomposing the seismic monitoring problem for heterogeneous processing

At the highest level, perfect parallelism is expected because real-time signals come in from monitors located at various sites using separate processors mostly running with Fortran seismic code.

Once the seismic signals are cleaned up by the distributed processors, we need to use a middle level of functional decomposition corresponding to a sequence of AI decisions.

Finally, the interpretation of real-time data may require domain decomposition for each signal domain.

Different types of computers are used at different levels in the layered approach. The monitor processors used at the top level are nothing but separated signal processors with numeric, Fortran, and vector processing capabilities.

The functional decomposition may use neural simulators, Lisp processors, or other symbolic machines with extended memory and intelligence interpretation capabilities.

The bottom level of domain decomposition demands a higher degree of parallelism if the survey sites are numerous. Graphics/visualization capabilities are needed to interpret the final results. The above example presents a typical case of heterogeneous processing in a network environment.

Summary

In this chapter, we studied software environments and program development techniques for parallel computers. Parallel programming languages must address important issues such as *compatibility*, *expressiveness*, *ease of use*, *efficiency*, and *portability*; and usually there are trade-offs involved while addressing these issues. Parallel programming environments must provide the required tools for program design, debugging, visualization, performance monitoring and tuning, input/output, and communication. As specific examples, we studied the Cray Y-MP, Paragon and CM-5 parallel programming environments.

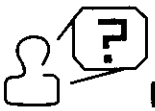
The closely related issues of synchronization and multiprocessing modes are central to any parallel processing environment. We studied the basic principles of synchronization in terms of *atomic operations*, *wait protocols*, *fairness policies*, and *sole-access protocols*. Multiprocessing requires fast context-switching and efficient synchronization. Multitasking may be seen as one variant of multiprocessing; depending on its granularity, it may be dubbed macro-tasking or micro-tasking. The multitasking mechanism provided on Cray multiprocessors was studied as a specific example.

One important model of parallel programs is the shared variable model. Since multiple parallel processes access shared variables in memory, locks can be employed for achieving protected access. A lock may be implemented as a *spin lock* or a *suspend lock*. Semaphores provide a higher level of synchronization mechanism than locks. The two basic operations on a semaphore s are $P(s)$ and $V(s)$. Monitors provide a still higher level of synchronization, encapsulating both shared variables and the permitted operations on them. We studied these synchronization mechanisms with the help of example applications.

Another equally important model of parallel programs is the message-passing model, in which the various processes running in parallel—on multiple processors—do not in general share main memory. Program development for multicomputers under this model must also address the issue of distributing the computation over available nodes, since this distribution determines the demands made on the message-passing subsystem, and thereby the overall system performance. In this context, the advantages

and disadvantages of synchronous versus asynchronous message passing were discussed, although it is true that the asynchronous model is more widely used.

Decomposition techniques are needed for mapping programs onto multicomputers. Domain decomposition and control decomposition techniques were described in this chapter, with several specific examples. The concept of heterogeneous processing was introduced.



Exercises

Problem 11.1 Explain the following terms associated with fast and efficient synchronization schemes on a shared-memory multiprocessor:

(a) Busy-wait versus sleep-wait protocols for sole access of a critical section.

(b) Fairness policies for reviving one of the suspended processes waiting in a queue.

(c) Lock mechanisms for pre-synchronization to achieve sole access to a critical section.

(d) Optimistic concurrency or the post-synchronization method.

(e) Server synchronization and the corresponding synchronization environment.

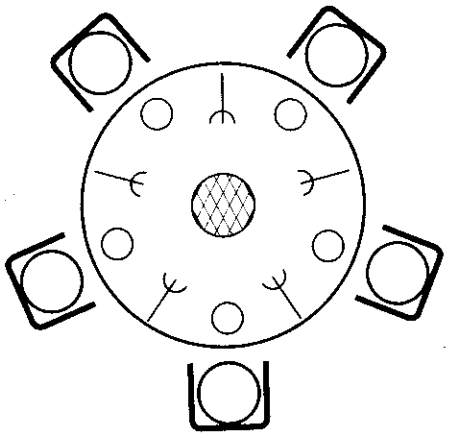
Problem 11.2 Distinguish between spin locks and suspend locks for sole access to a critical section. Generalize Dekker's protocol from two procedures to three or more procedures sharing critical sections. Also implement the generalized Dekker's protocol using the Test&Set atomic operation.

Problem 11.3 There are many ways to solve the mutual exclusion problem based on different implementation schemes, such as the use of spin locks or Dekker's protocol. Describe the following schemes:

(a) Implementing mutually exclusive access to a critical section using *binary semaphores*.

(b) Implementing mutual exclusion using a *monitor* called by processes competing for access to a critical section.

Problem 11.4 Dijkstra [Dijkstra68] has defined the well-known dining philosophers problem: There are five philosophers dining around a table as shown.



Each of the philosophers engages in only two activities, *thinking* or *eating*, as characterized below as process P_i for $i = 1, 2, \ldots, 5$.

```
Pi: begin
      loop
        Think
        Fetch protocol
        Eat
        Release protocol
      endloop
    end
```

A bowl with an endless supply of spaghetti is placed in the center of the table. Each philosopher is given a plate. There are five forks on the table, one between adjacent philosophers. Each philosopher enters a thinking period while not eating. In order to be able to eat, the philosopher must get hold of two neighboring forks from his left and right sides. The *fetch protocol* specifies how each of the forks will be picked up. The *release protocol* specifies how the forks will be released after eating. Each philosopher is allowed to pick up one fork at a time from his left or from his right side. Each fork can be used by only one philosopher at a time. No fork can be passed around the table, and a fork must be put back where it was picked up.

The problem is to design the fetch protocol and the release protocol for the philosophers so that no deadlock will occur. A deadlock means a circular wait situation in which each philosopher holds one fork and refuses to release it. A deadlock means starvation, so we should avoid it in the solution.

(a) Use the *P* and *V* operators and binary semaphores to specify the fetch and release protocols. One semaphore can be used to represent the fork on the right, and another semaphore to represent the fork on the left. The purpose of the protocols is to prevent deadlock from occurring so that no individual starvation will occur. Initially, all the forks are placed on the table corresponding to an initial value of 1 for the semaphores. When a fork is picked up, its semaphore is changed to a value of 0.

(b) Design a *monitor* to control the fetch and release of the forks. In this case, the fetch and release are each specified by a procedure in the monitor. Also, specify the philosophers as user processes calling the monitor to claim forks. Again, no deadlock or starvation is allowed under the same assumption made in part (a).

Problem 11.5 Explain why *mutual exclusion*, *nonpreemption*, *wait for*, and *circular wait* are necessary conditions but not sufficient conditions for a system deadlock to occur. Also distinguish among the *deadlock prevention*, *avoidance*, *detection*, and *recovery* schemes. Comment on their implementation costs and expected performances.

Problem 11.6 Five concurrent processes are specified below using four resource types represented by four semaphores. Answer the following questions with reasoning and justification.

```
Begin
   shared record
      begin
         var S1, S2, S3, S4: semaphore;
         var blocked, unblocked: integer;
      end
   initial blocked = 0, unblocked = 1;
   initial S1= S2 = S3 = unblocked; S4 = blocked;
   cobegin
      A: begin P(S1);V(S1); P(S2);V(S2); end;
      B: begin P(S1); P(S2);V(S4);V(S2);V(S1); end;
      C: begin P(S2); P(S3);V(S2);V(S3); end;
      D: begin P(S4); P(S2); P(S1);V(S1);V(S2); end;
      E: begin P(S3); P(S2); V(S2);V(S3); end;
   coend
End
```

(a) Is a deadlock possible among the five code segments represented by A, B, C, D, and E? Which subset of code segments may enter a deadlock on what resources?

(b) If the deadlock situation does occur in part (a), what additional code segments could be indefinitely blocked?

(c) Is a deadlock inevitable or does it depend on race conditions? Justify your answer with reasoning using a resource allocation graph.

(d) Make a minor change in one program segment

to prevent a deadlock from occurring. Justify the claim with a resource allocation graph similar to Fig. 11.5.

Problem 11.7 Scheduling access to a moving-head disk can be implemented by a monitor. The implementation consists of three components: *user processes* which request, access, and release the disk service; a *disk scheduler* which performs the scheduling of disk data to be accessed by one user at a time; and *driver procedures* that perform actual data transfer.

(a) Write a *monitor* to implement the disk scheduler. The monitor should consist of two procedures, one for a request for and one for a release from disk access.

(b) Specify how a user process can call the monitor for disk access. The disk driver procedures are considered given.

Problem 11.8 Write a monitor as a barrier counter for the synchronization of n concurrent processes. The barrier counter should be resettable, and a user process should be specified to call the monitor when it reaches the barrier. Note that local and shared variables must be declared and initialization of local data must be given.

Problem 11.9 Answer the following questions on decomposition techniques for message-passing programming on multicomputer nodes:

(a) What is a perfect decomposition? Explain the advantages and discuss the differences in program replication techniques on multicomputers as opposed to program partitioning on multiprocessors.

(b) Based on data domain, algorithm used, and flow of control in applications, distinguish the opportunities for applying domain, control, and object decomposition techniques in distributed computing on multicomputers.

Problem 11.10 The *N-queens problem* (Fig. 11.10) was introduced along with the manager-worker approach to control decomposition in programming a multicomputer. Suppose $N = 8$. There are 92 possible solutions to the 8-queens problem.

(a) Write a program that searches for a solution. First run the program on a sequential computer (such as on a workstation or even a personal computer). Record the time required to conduct the sequential search. A sequential search involves backtracking once an impossible configuration is exposed. The backtracking step systematically removes the current emplacements of the queens and then continues with a new emplacement.

(b) Develop a parallel program to run on a message-passing multicomputer if one is available. For a concurrent search for solutions to the N-queens problem, backtracking is not necessary because all solutions are equally pursued. A detected impossible configuration is simply discarded by the node. Observe the dynamics of the concurrent search activities and record the total execution time. Compare the measured execution time data and comment on speedup gain and other performance issues.

Problem 11.11 The *traveling salesperson problem* is to find the shortest route connecting a set of cities, visiting each city only once. The difficulty is that as the number of cities grows, the number of possible paths connecting them grows exponentially. In fact, $(n - 1)!/2$ paths are possible for n cities. A parallel program, based on *simulated annealing*, was developed by Caltech researchers Felten, Karlin, and Otto [Felten85] for solving the problem for 64 cities grouped in 4 clusters of 16 each on a multicomputer.

(a) Kallstrom and Thakkar (1988) implemented the Caltech program in C language on an iPSC/1 hypercube computer with 32 nodes. Study this C program for solving the traveling salesperson problem using a simulated annealing technique. Describe the concurrency opportunities in performing the large number of iterations (such as 60,000) per temperature drop.

(b) Rerun the code on a modern message-passing multicomputer. Check the execution time and performance results and compare them with those reported by Kallstrom and Thakkar. You will need to modify the code in order to run on a different machine.

Problem 11.12 Choose an example program to demonstrate the concepts of macrotasking, microtasking, and autotasking on a Cray-like multiprocessor supercomputer. Perform a tradeoff study on the relative performance of the three multitasking schemes based on the example program execution. Make reasonable assumptions as needed, as in Example 11.1.

Problem 11.13 Write a multitasked vectorized code in Fortran 90 for matrix multiplication using four processors with a shared memory. Assume square matrices of order $n = 4k$. The entire data set is available from the shared memory.

Problem 11.14 Design a message-passing program for performing fast Fourier transform (FFT) over 1024 sample points on a 32-node hypercube computer. Both host and node programs should be specified, including all communication commands. Initially each node holds 32 sample points without duplicating the data set. The results should be sent to the host for output.

Problem 11.15 A typical two-dimensional image is represented by a rectangular array of pixels (picture elements). Each pixel (i, j) is represented by a $\log_2 b$ bit integer corresponding to the gray level (between 0 and $b - 1$) at coordinate (i, j) of a black-and-white picture. *Histogramming* is a process to count the frequency of occurrences of each gray level. Let histog$(0 : b - 1)$ be the array of a histogram of b gray levels.

The following serial code is written for histogramming on a uniprocessor system:

```
Var pixel(0: m − 1, 0: n − 1);
Var histog(0 : b − 1): integer;
histog(0: b − 1) = 0;
for i = 0 to m − 1 do
    for j = 0 to n − 1 do
        histog(pixel(i,j)) = histog(pixel(i,j)) + 1
```

The time complexity (number of counts) of this serial program is $O(mn)$, where mn corresponds to the image size.

Partition the image, pixel(i, j) for $0 \le i \le m - 1$ and $0 \le j \le n - 1$ into p disjoint segments, where each segment has $m/p = s$ rows of the image. Develop a parallel program which can spawn a set of p processes to histogram the entire image. The p concurrent processes share the same histogram array histog$(0 : b - 1)$.

(a) Use the **Doall** and **Endall** statements to specify a parallel program for counting the histogram simultaneously on a p-processor system with shared memory.

(b) What is the potential speedup of the parallel program over the above serial program? You can ignore the image I/O overhead by assuming the entire image database is in the main memory.

Part V

Instruction and System Level Parallelism

Chapter 12

Instruction Level Parallelism

Chapter 13

Trends in Parallel Systems

Summary

The basic concepts of parallel computer systems—theoretical formulations, hardware architecture, and programming models—have been discussed in detail in Parts I through IV of the book. The decades of 1970s and 1980s generated a great many innovative ideas in computer architecture. Since then, over the last couple of decades, the technologies underlying computer architecture—VLSI, storage, interconnects, and so on—have seen huge advances, and these have had a huge impact on computer architecture. At the same time, the range of applications of computer systems has also grown enormously.

Against this background, Chapter 12 discusses the important topic of *instruction level parallelism* (ILP), which has a crucial bearing on processor design. We see that the issue of exploitation of ILP is a system design issue, and we also discuss the limitations which are encountered in exploiting ILP in real-life applications.

Chapter 13 discusses recent trends in parallel computer systems—for this, however, it is also necessary to discuss in brief the technological advances which have impacted computer architecture. Some basic concepts related to parallel algorithms are discussed, and a number of case studies are presented of processors, systems-on-a-chip, and massively parallel systems. The parallel programming language Chapel is introduced, as also function libraries which have been developed for writing parallel programs.

12

Instruction Level Parallelism

12.1 INTRODUCTION

The period between the 1970s and the 1990s saw a great many innovative ideas being proposed in computer architecture. The basic hardware technology of computers had been mastered by the 1960s, and several companies had produced successful commercial products. The time was therefore right to generate new ideas, to reach performance levels higher than that of the original single-processor systems. As we have seen, parallelism in its various forms has played a central role in the development of newer architectures.

The earlier part of this book has presented a comprehensive overview of the many architectural innovations which had been attempted until the early 1990s. Some of these were commercially successful, while many others were not so fortunate—which is not at all surprising, given the large variety of ideas which were proposed and the fast-paced advances taking place in the underlying technologies.

In the last two chapters of the book, we take a look at some of the recent trends and developments in computer architecture—including, as appropriate, a brief discussion of advances in the underlying technologies which have made these developments possible. In fact, we shall see that the recent advances in computer architecture can be understood only when we also take a look at the underlying technologies.

What is computer architecture?

(a) We define *computer architecture* as the arrangement by which the various system building blocks—processors, functional units, main memory, cache, data paths, and so on—are interconnected and inter-operated to achieve desired *system performance*.

(b) Processors make up the most important part of a computer system. Therefore, in addition to (a), *processor design* also constitutes a central and very important element of computer architecture. Various functional elements of a processor must be designed, interconnected and inter-operated to achieve desired *processor performance*.

System performance is the key benchmark in the study of computer architecture. A computer system must solve the real world problem, or support the real world application, for which the user is installing it. Therefore, in addition to the theoretical peak performance of the processor, the design objectives of any computer architecture must also include other important criteria, which include system performance under

realistic load conditions, scalability, price, usability, and reliability. In addition, power consumption and physical size are also often important criteria.

A basic rule of system design is that *there should be no performance bottlenecks in the system.* Typically, a performance bottleneck arises when one part of the system—i.e. one of its subsystems—cannot keep up with the overall throughput requirements of the system. Such a performance bottleneck can occur in a production system, a distribution system, or even in traffic system[1]. If a performance bottleneck does occur in a system—i.e. if one subsystem is not able to keep up with other subsystems—then the other subsystems remain idle, waiting for response from the slower one.

In a computer system, the key subsystems are processors, memories, I/O interfaces, and the data paths connecting them. Within the processors, we have subsystems such as functional units, registers, cache memories, and internal data buses. Within the computer system as a whole—or within a single processor—designers do not wish to create bottlenecks to system performance.

Example 12.1 Performance bottleneck in a system

In Fig. 12.1 we see the schematic diagram of a simple computer system consisting of four processors, a large shared main memory, and a processor-memory bus.

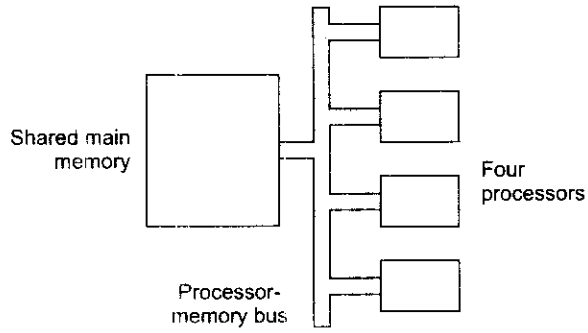


Fig. 12.1 A simple shared memory multiprocessor system

For the three subsystems, we assume the following performance figures:

(i) Each of the four processors can perform double precision floating point operations at the rate of 500 million per second, i.e. 500 MFLOPs.

(ii) The shared main memory can read/write data at the aggregate rate of 1000 million 32-bit words per second.

(iii) The processor-memory bus has the capability of transferring 500 million 32-bit words per second to/from main memory.

[1] In common language, we say that *a chain is only as strong as its weakest link.*

This system exhibits a performance mismatch between the processors, main memory, and the processor-memory bus. The data transfer rates supported by the main memory and the shared processor-memory bus do not meet the aggregate requirements of the four processors in the system.

The system architect must pay careful attention to all such potential mismatches in system design. Otherwise, the sustained performance which the system can deliver can only equal the performance of the slowest part of the system—i.e. the bottleneck.

While this is a simple example, it illustrates the key challenge facing system designers. It is clear that, in the above system, if *processor performance* is improved by, say, 20%, we may not see a matching improvement in *system performance*, because the performance bottleneck in the system is the relatively slower processor-memory bus. In this particular case, a better investment for increased system performance could be (a) faster processor-memory bus, and (b) improved cache memory with each processor, i.e. one with better hit rate—which reduces contention for the processor-memory bus.

In fact, as we shall see, even achieving peak theoretical performance is not the final goal of system design. The system performance must be maintained for real-life applications, and that too in spite of the enormous diversity in modern applications.

In earlier chapters of the book, we have studied the many ways in which parallelism can be introduced in a computer system, for higher processing performance. The concept of instruction level parallelism and superscalar architecture has been introduced in Chapter 6. In this chapter, we take a more detailed look at instruction level parallelism.

12.2 BASIC DESIGN ISSUES

As we have seen in Chapter 6, a linear instruction pipeline is the basic structure which exploits instruction level parallelism in the executing sequence of machine instructions. We have also discussed in brief how further hardware techniques can be employed with a view to achieve *superscalar* processor architecture—i.e. multiple instruction issues in every processor clock cycle. In this chapter, we shall study these and other related concepts in some more detail.

Instruction pipeline and cache memory (or multi-level cache memories) hide the memory access latencies of instruction execution. With multiple functional units within the processor, *superscalar* instruction execution rates—greater than one per processor clock cycle—can be targeted, using multiple issue pipeline architecture. The aim is that the enormous processing power made possible by VLSI technology must be utilized to the full, ideally with each functional unit producing a result in every clock cycle. For this, the processor must also have data paths of requisite bandwidth—within the processor, to the memory and I/O subsystems, and to other processors in a multiprocessor system.

With a single processor chip today containing a billion (10^9) or more transistors, system design is not possible in the absence of a target application. For example, is a processor being designed for intensive scientific number-crunching, a commercial server, or for desktop applications?

One key design choice which appears in such contexts is the following.

Should the primary design emphasis be on:

(a) exploiting fully the parallelism present in a single instruction stream, or

(b) supporting multiple instruction streams on the processor in multi-core and/or multi-threading mode?

This design choice is also related to the depth of the instruction pipeline. In general, designs which aim to maximize the exploitation of instruction level parallelism need deeper pipelines; up to a point, such designs may support higher clock rates. But, beyond a point, deeper pipelines do not necessarily provide higher net throughput, while power consumption rises rapidly with clock rate, as we shall also discuss in Chapter 13.

Let us examine the trade-off involved in this context in a simplified way:

$$\text{total chip area} = \text{number of cores} \times \text{chip area per core}$$

or

$$\text{total transistor count} = \text{number of cores} \times \text{transistor count per core}$$

Here we have assumed for simplicity that cache and interconnect area—and transistor count—can be considered proportionately on a per core basis.

At a given time, VLSI technology limits the left hand side in the above equations, while the designer must select the two factors on the right. Aggressive exploitation of instruction level parallelism, with multiple functional units and more complex control logic, increases the chip area—and transistor count—per processor core. Alternatively, for a different category of target applications, the designer may select simpler cores, and thereby place a larger number of them on a single chip.

Of course system design would involve issues which are more complex than these, but a basic design issue is seen here: For the targeted application and performance, how should the designers divide available chip resources among processors and, within a single processor, among its various functional elements?

Within a processor, a set of instructions are in various stages of execution at a given time—within the pipeline stages, functional units, operation buffers, reservation stations, and so on. Recall that functional units themselves may also be internally pipelined. Therefore machine instructions are not in general executed in the order in which they are stored in memory, and all instructions under execution must be seen as 'work in progress'.

As we shall see, to maintain the work flow of instructions within the processor, a superscalar processor makes use of *branch prediction*—i.e. the result of a conditional branch instruction is predicted even before the instruction executes—so that instructions from the predicted branch can continue to be processed, without causing pipeline stalls. The strategy works provided fairly good branch prediction accuracy is maintained.

But we shall assume that instructions are *committed* in order. Here *committing* an instruction means that the instruction is no longer 'under execution'—the processor state and program state reflect the completion of all operations specified in the instruction.

Thus we assume that, at any time, the set of committed instructions correspond with the program order of instructions and the conditional branches actually taken. Any hardware exceptions generated within the processor must reflect the processor and program state resulting from instructions which have already committed.

Parallelism which appears explicitly in the source program, which may be dubbed as *structural parallelism*, is not directly related to instruction level parallelism. Parallelism detected and exploited by the compiler is a form of instruction level parallelism, because the compiler generates the machine instructions which result in parallel execution of multiple operations within the processor. We shall discuss in Section 12.5 some of the main issues related to this method of exploiting instruction level parallelism.

Parallelism detected and exploited by processor hardware *on the fly*, within the instructions which are under execution, is certainly instruction level parallelism. Much of the remaining part of this chapter discusses the basic techniques for hardware detection and exploitation of such parallelism, as well as some related design trade-offs.

While the student is expected to be familiar with the basic concepts related to instruction pipelines, the earlier discussion of these topics in Chapter 6 will serve as an introduction to the techniques discussed more fully in this chapter.

Weak memory consistency models, which are discussed elsewhere in the book, are not discussed explicitly in this chapter, since they are relevant mainly in the case of parallel threads of execution distributed over multiple processors. Similarly—since the discussion in this chapter is primarily in the context of a single processor—the issues of shared memory, cache coherence, and message-routing are also not discussed here. The student may refer to Chapters 5 and 7, respectively, for a discussion of these two topics.

With this background, let us start with a statement of the basic system design objective which is addressed in this chapter.

12.3 PROBLEM DEFINITION

Let us now focus our attention on the execution of machine instructions from a single sequential stream. The instructions are stored in main memory in program order, from where they must be fetched into the processor, decoded, executed, and then committed in program order. In this context, we must address the problem of detecting and exploiting the parallelism which is implicit within the instruction stream.

We need a prototype instruction for our processor. We assume that the processor has a *load-store* type of instruction set, which means that all arithmetic and logical operations are carried out on operands which are present in programmable registers. Operands are transferred between main memory and registers by *load* and *store* instructions only.

We assume a three-address instruction format, as seen on most RISC processors, so that a typical instruction for arithmetic or logical operation has the format:

opcode operand-1 operand-2 result

Our aim is to make the discussion independent of any specific instruction set, and therefore we shall use simple and self-explanatory opcodes, as needed.

Data transfer instructions have only two operands—source and destination registers; *load* and *store* instructions to/from main memory specify one operand in the form of a memory address, using an available addressing mode. Effective address for *load* and *store* is calculated at the time of instruction execution.

Conditional branch instructions need to be treated as a special category, since each such branch presents two possible continuations of the instruction stream. Branch decision is made only when the instruction executes; at that time, if instructions from the branch-not-taken are in the pipeline, they must be *flushed*. But pipeline flushes are costly in terms of lost processor clock cycles. The payoff of branch prediction lies in the fact that correctly predicted branches allow the detection of parallelism to stretch across two or more basic

blocks of the program, without pipeline stalls. It is for this reason that branch prediction becomes an essential technique in exploiting instruction level parallelism.

Limits to detecting and exploiting instruction level parallelism are imposed by *dependences* between instructions. After all, if N instructions are completely independent of each other, they can be executed in parallel on N functional units—if N functional units are available—and they may even be executed in arbitrary order.

But in fact dependences amongst instructions are a central and essential part of program logic. A dependence specifies that instruction I_k must wait for instruction I_j to complete. Within the instruction pipeline, such a dependence may create a *hazard* or *stall*—i.e. lost processor clock cycles while I_k waits for I_j to complete.

For this reason, for a given instruction pipeline design and associated functional units, dependences amongst instructions limit the available instruction level parallelism—and therefore it is natural that the central issue in exploiting instruction level parallelism is related to the correct handling of such dependences.

We have already seen in Chapter 2 that dependences amongst instructions fall into several categories; here we shall review these basic concepts and introduce some related notation which will prove useful.

Data Dependences

Assume that instruction I_k follows instruction I_j in the program. *Data dependence* between I_j and I_k means that both access a common operand. For the present discussion, let us assume that the common operand of I_j and I_k is in a programmable register. Since each instruction either reads or writes an operand value, accesses by I_j and I_k to the common register can occur in one of four possible ways:

Read by I_k after read by I_j
Read by I_k after write by I_j
Write by I_k after read by I_j
Write by I_k after write by I_j

Of these, the first pattern of register access does not in fact create a dependence, since the two instructions can read the common value of the operand in any order.

The other three patterns of operand access do create dependences amongst instructions. Based on the underlined words shown above, these are known as *read after write* (RAW) dependence, *write after read* (WAR) dependence, and *write after write* (WAW) dependence, respectively.

Read after write (RAW) is true data dependence, in the sense that the register value written by instruction I_j is read—i.e. used—by instruction I_k. This is how computations proceed; a value produced in one step is used further in a subsequent step. Therefore RAW dependences must be respected when program instructions are executed. This type of dependence is also known as *flow dependence*.

Write after read (WAR) is known as *anti-dependence*, because in this instance instruction I_k should not overwrite the value in the common register until the previous value stored therein has been used by the prior instruction I_j which needs the value. Such dependence can be removed from the executing program by simply assigning another register for the write instruction I_k to write into. With read and write occurring to two different registers, the dependence between instructions is removed. In fact, this is the basis of the *register renaming* technique which we shall discuss later in this chapter.

Write after write (WAW) is known as *output dependence*, since two instructions are writing to a common register. If this dependence is violated, then subsequent instructions will see a value in the register which should in fact have been overwritten—i.e. they will see the value written by I_j rather than I_k. This type of dependence can also be removed from the executing program by assigning another target register for the second write instruction, i.e. by *register renaming*.

Sometimes we need to show dependences between instructions using graphical notation. We shall use small circles to represent instructions, and double line arrows between two circles to denote dependences. The instruction at the head of the arrow is dependent on the instruction at the tail; if necessary, the type of dependence between instructions may be shown by appropriate notation next to the arrow. A missing arrow between two instructions will mean explicit absence of dependence.

Single line arrows will be used between instructions when we wish to denote program order without any implied dependence or absence of dependence.

Figure 12.2 illustrates this notation.

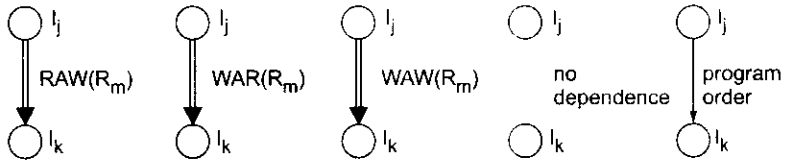


Fig. 12.2 Dependences shown in graphical notation (R_m indicates register)

When dependences between multiple instructions are thus depicted, the result is a *directed graph* of dependences. A *node* in the graph represents an instruction, while a *directed edge* between two nodes represents a dependence.

Often dependences are thus depicted in a *basic block* of instructions— i.e. a sequence of instructions with entry only at the first instruction, and exit only at the last instruction of the sequence. In such cases, the graph of dependences becomes a *directed acyclic graph*, and the dependences define a *partial order* amongst the instructions.

Part (a) of Fig. 12.3 shows a *basic block* of six instructions, denoted I_1 through I_6 in program order. Entry to the basic block may be from one of multiple points within the program: continuation after the basic block would be at one of several points, depending on the outcome of conditional branch instruction at the end of the block.

Part (b) of the figure shows a possible pattern of dependences as they may exist amongst these six instructions. For simplicity, we have not shown the type of each dependence, e.g. RAW(R_3), etc. In the partial order, we see that several pairs of instructions--such as (I_1, I_2) and (I_3, I_4)--are not related by any dependence. Therefore, amongst each of these pairs, the instructions may be executed in any order, or in parallel.

Dependences amongst instructions are inherent in the instruction stream. For processor design, the important questions are: For a given processor architecture, what is the effect of such dependences on processor performance? Do these dependences create hazards which necessitate pipeline stalls and/or flushes? Can these dependences be removed *on the fly* using some design technique? Can their adverse impact be reduced?

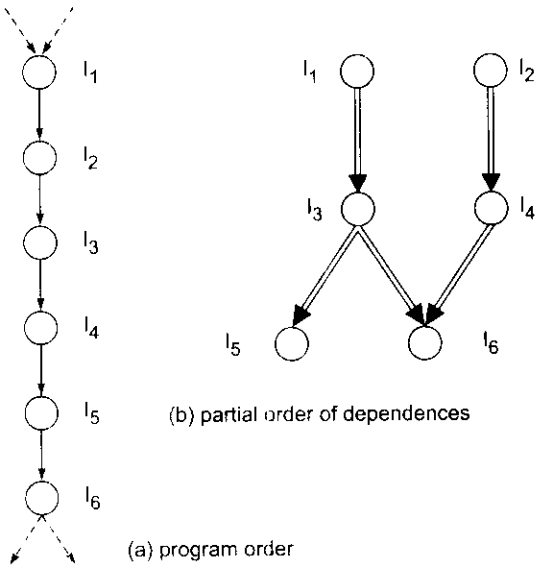


Fig. 12.3 A basic block of six instructions

Consider once again the pattern of dependences shown in Fig. 12.3(b). If the processor is capable of completing two (or more) instructions per clock cycle, and if no pipeline stalls are caused by the dependences shown, then clearly the six instructions can be completed in three consecutive processor clock cycles. Instruction latency, from fetch to commit stage, will of course depend on the depth of the pipeline.

Control Dependences In typical application programs, *basic blocks* tend to be small in length, since about 15% to 20% instructions in programs are branch and jump instructions, with indirect jumps and *returns* from procedure calls also included in the latter category. Because of typically small sizes of basic blocks in programs, the amount of instruction level parallelism which can be exploited in a single basic block is limited.

Assume that instruction I_j is a conditional branch and that, whether another instruction I_k executes or not depends on the outcome of the conditional branch instruction I_j. In such a case, we say that there is a *control dependence* of instruction I_k on instruction I_j.

Let us assume that a processor has instruction pipeline of depth eight, and that the designers target superscalar performance of four instructions completed in every clock cycle. Assuming no pipeline stalls, the number of instructions in the processor at any one time—in its various pipeline stages and functional units—would be $4 \times 8 = 32$.

If 15% to 20% of these instructions are branches and jumps, then the execution of subsequent instructions within the processor would be held up pending the resolution of conditional branches, procedure returns, and so on—causing frequent pipeline stalls.

This simple calculation shows the potential adverse impact of conditional branches on the performance of a superscalar processor. The key question here is: How can the processor designer mitigate the adverse impact of such *control dependences* in a program?

Answer: Using some form of *branch and jump prediction*—i.e. predicting early and correctly (most of the time) the results of conditional branches, indirect jumps, and procedure returns. The aim is that, for every correct prediction made, there should be no lost processor clock cycles due to the conditional branch, indirect jump, or procedure return. For every mis-prediction made, there would be the cost of flushing the pipeline of instructions from the wrong continuation after the conditional branch or jump.

Example 12.2 Impact of successful branch prediction

Assume that we have attained 93% accuracy in branch prediction in a processor with eight pipeline stages. Assume also that the mis-prediction penalty is 4 processor clock cycles to flush the instruction pipeline. What is the performance gain from such a branch prediction strategy?

Recall that the expected cost of a random variable X is given by $\Sigma x_i p_i$, where x_i are possible values of X, and p_i are the respective probabilities. In our case, the probability of a correct branch is 0.93, and the corresponding cost is zero; the probability of a wrong branch is 0.07, and the corresponding cost is 2. Thus the expected cost of a conditional branch instruction is $0.07 \times 4 = 0.28$ clock cycle i.e. much less than one clock cycle.

As a primitive form of branch prediction, the processor designer could assume that a conditional branch is always taken, and continue processing the instructions which follow at the target address. Let us assume that this simple strategy works 80% of the time; then the expected cost of a conditional branch is $0.2 \times 4 = 0.8$ clock cycles.

Suppose that not even this primitive form of branch prediction is used Then the pipeline must stall until the result of every branch condition, and the target address of every indirect jump and procedure return, is known; only then can the processor proceed with the correct continuation within the program. If we assume that in this case the pipeline stalls over half the total number of stages, then the number of lost clock cycles is 4 for every conditional branch, indirect jump and procedure return instruction.

Considering that 15% to 20% of the instructions in a program are branches and jumps, the difference in cost between 0.28 clock cycle and 4 clock cycles per branch instruction is huge, underlining the importance of branch prediction in a superscalar processor.

Later, in this chapter, we shall study the techniques employed for branch prediction.

Resource Dependences This is possibly the simplest kind of dependence to understand, since it refers to a resource constraint causing dependence amongst instructions needing the resource.

Example 12.3 Resource dependence

Consider a simple pipelined processor with only one floating point multiplier, which is not internally pipelined and takes three processor clock cycles for each multiplication. Assume that several independent floating point multiply instructions follow each other in the instruction stream in a single basic block under execution.

Clearly, while the processor is executing these multiply instructions, it cannot for that duration get even one instruction completed in every clock cycle. Therefore pipeline stalls are inevitable, caused by the absence of sufficient floating point multiply capability within the processor. In fact, for the duration of these consecutive multiply operations, the processor will only complete one instruction in every three clock cycles.

We have assumed the instructions to be independent of each other, and in a single basic block—i.e. there are no conditional branches within the sequence. Thus there is no data dependence or control dependence amongst these instructions. What we have here is *resource dependence*, i.e. all the instructions depend on the resource which has not been provided to the extent it is needed for the given workload on the processor.

We can say that there is an imbalance in this processor between the floating point capability provided and the workload which is placed on it. Such imbalances in system resources usually have adverse performance impact. Recall that Example 12.1 above and the related discussion illustrated this same point in another context.

A *resource dependence* which results in a pipeline stall can arise for access to any processor resource—functional unit, data path, register bank, and so on[2]. We can certainly say that such resource dependences will arise if hardware resources provided on the processor do not match the needs of the executing program.

Now that we have seen the various types of dependences which can occur between instructions in an executing program, the problem of detecting and exploiting instruction level parallelism can finally be stated in the following manner:

Problem Definition Design a superscalar processor to detect and exploit the maximum degree of parallelism available in the instruction stream—i.e. execute the instructions in the smallest possible number of processor clock cycles—by handling correctly the data dependences, control dependences and resource dependences within the instruction stream.

Before we can make progress in that direction, however, it is necessary to keep in mind a prototype processor design on which the problem solution can be attempted.

12.4 MODEL OF A TYPICAL PROCESSOR

We assume a processor with *load-store* instruction set architecture and a set of programmable registers as seen by the assembly language programmer or the code generator of a compiler. Whether these registers are bifurcated into separate sets of integer and floating point registers is not important for us at present, nor is the exact number of these registers.

To support parallel access to instructions and data at the level of the fastest cache, we assume that L1 cache is divided into instruction cache and data cache, and that this split L1 cache supports single cycle access for instructions as well as data. Some processors may have an *instruction buffer* in place of L1 instruction cache; for the purposes of this section, however, the difference between them is not important.

The first three pipeline stages on our prototype processor are *fetch*, *decode* and *issue*.

Following these are the various functional units of the processor, which include integer unit(s), floating point unit(s), load/store unit(s), and other units as may be needed for a specific design—as we shall see when we discuss specific design techniques.

---

[2] This type of dependence may also be called *structural dependence*, since it is related to the structure of the processor; however *resource dependence* is the more common term.

Let us assume that our superscalar processor is designed for k instruction issues in every processor clock cycle. Clearly then the *fetch*, *decode* and *issue* pipeline stages, as well as the other elements of the processor, must all be designed to process k instructions in every clock cycle.

On multiple issue pipelines, *issue* stage is usually separated from *decode* stage. One reason for thus increasing a pipeline stage is that it allows the processor to be driven by a faster clock. *Decode* stage must be seen as preparation for instruction *issue* which—by definition—can occur only if the relevant functional unit in the processor is in a state in which it can accept one more operation for execution. As a result of the *issue*, the operation is handed over to the functional unit for execution.

Note 12.1

The name of instruction *decode* stage is somewhat inaccurate, in the sense that the instruction is never fully decoded. If a 32-bit instruction is fully decoded, for example, the decoder would have some 4×10^9 outputs! This is never done; an immediate constant is never decoded, and memory or I/O address is decoded outside the processor, in the address decoder associated with the memory or I/O module.

Register select bits in the instruction are decoded when they are used to access the register bank; similarly, ALU function bits can be decoded within the ALU. Therefore register select and ALU function bits also need not be decoded in the instruction *decode* stage of the processor.

What happens in the instruction *decode* stage of the processor is that some of the key fields of the instruction are decoded. For example, opcode bits must be decoded to select the functional unit, and addressing mode bits must be decoded to determine the operations required to calculate effective memory address.

The process of issuing instructions to functional units also involves *instruction scheduling*[3]. For example, if instruction I_j cannot be issued because the required functional unit is not free, then it may still be possible to issue the next instruction I_{j+1}—provided that no dependence between the two prohibits issuing instruction I_{j+1}.

When instruction scheduling is specified by the compiler in the machine code it generates, we refer to it as *static scheduling*. In theory, static scheduling should free up the processor hardware from the complexities of instruction scheduling; in practice, though, things do not quite turn out that way, as we shall see in the next section.

If the processor control logic schedules instruction *on the fly*—taking into account inter-instruction dependences as well as the state of the functional units—we refer to it as *dynamic scheduling*. Much of the rest of this chapter is devoted to various aspects and techniques of dynamic scheduling. Of course the basic aim in both types of scheduling—static as well as dynamic—is to maximize the instruction level parallelism which is exploited in the executing sequence of instructions.

As we have seen, at one time multiple instructions are in various stages of execution within the processor. But *processor state* and *program state* need to be maintained which are consistent with the program order of completed instructions. This is important from the point of view of preserving the semantics of the program.

Therefore, even with multiple instructions executing in parallel, the processor must arrange the results of completed instructions so that their sequence reflects program order. One way to achieve this is by using a

[3] Instruction scheduling as discussed here has some similarity with other types of task or job scheduling systems. It should be noted, of course, that a typical production system requiring job scheduling does not involve conditional branches, i.e. control dependences.

reorder buffer, shown in Fig. 12.4, which allows instructions to be *committed* in program order, even if they execute in a different order; we shall discuss this point in some more detail in Section 12.7.

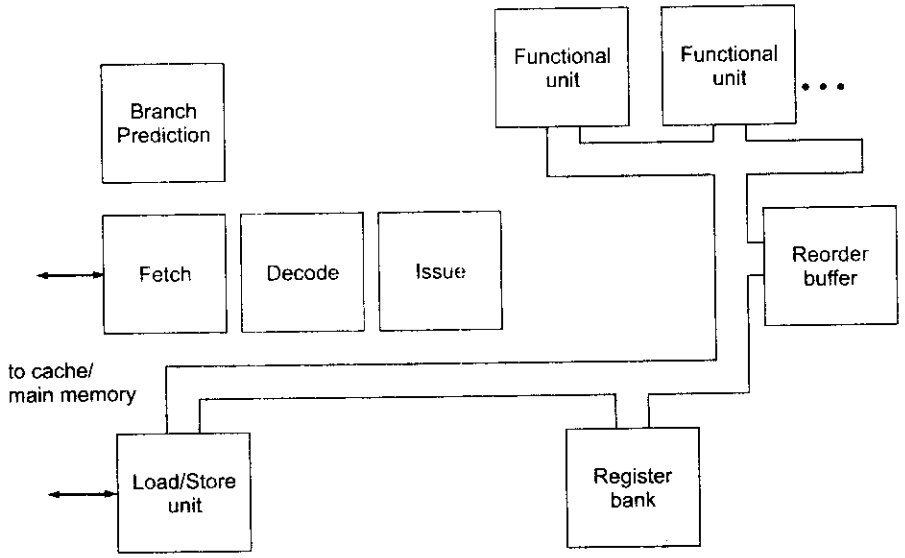


Fig. 12.4 Processor design with reorder buffer

If instructions are executed on the basis of predicted branches, before the actual branch outcome is available, we say that the processor performs *speculative execution*. In such cases, the reorder buffer will need to be cleared—wholly or partly—if the actual branch result indicates that speculation has occurred on the basis of a mis-prediction.

Functional units in the processor may themselves be internally pipelined; they may also be provided with *reservation stations*, which accept operations issued by the *issue* stage of the instruction pipeline. A functional unit performs an operation when the required operands for it are available in the reservation station. For the purposes of our discussion, memory *load-store unit(s)* may also be treated as functional units, which perform their functions with respect to the cache/memory subsystem.

Figure 12.5 shows a processor design in which functional units are provided with *reservation stations*. Such designs usually also make use of *operand forwarding* over a *common data bus* (CDB), with *tags* to identify the source of data on the bus. Such a design also implies *register renaming*, which resolves RAW and WAW dependences. Dynamic scheduling of instructions on such a processor is discussed in some more detail in Sections 12.8 and 12.9.

A branch prediction unit has also been shown in Fig. 12.4 and Fig. 12.5 to implement some form of a branch prediction algorithm, as discussed in Section 12.10.

Data paths connecting the various elements within the processor must be provided so that no *resource dependences*—and consequent pipeline stalls—are created for want of a data path. If k instructions are to be completed in every processor clock cycle, the data paths within the processor must support the required data transfers in each clock cycle.

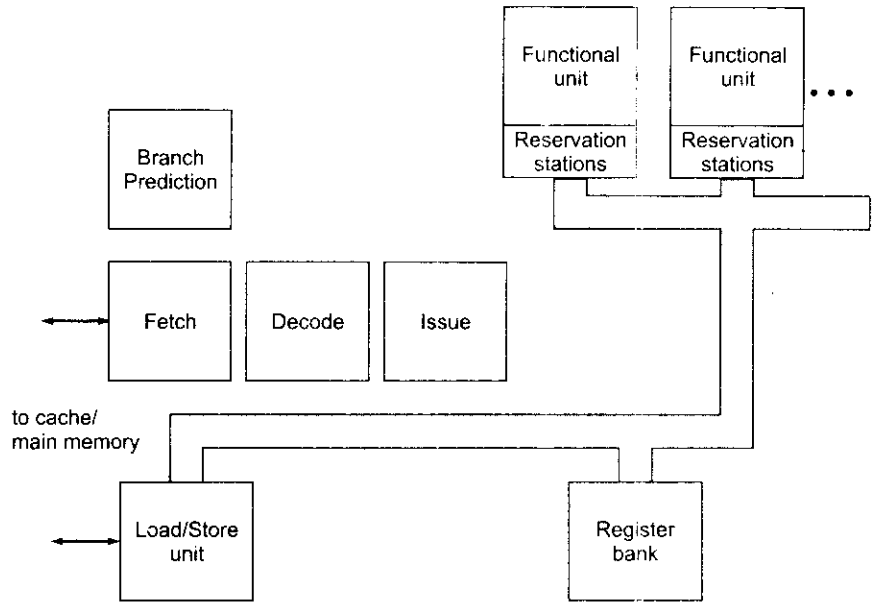


Fig. 12.5 Processor design with reservation stations on functional units

At one extreme, a primitive arrangement would be to provide a single common bus within the processor; but such a bus would become a scarce and performance limiting resource amongst multiple instructions executing in parallel within the processor.

At the other extreme, one can envisage a *complete graph* of data paths amongst the various processor elements. In such a system, in each clock cycle, any processor element can transfer data to any other processor element, with no resource dependences caused on that account. But unfortunately, for a processor with n internal elements, such a system requires $n - 1$ data ports at every element, and is therefore not practical.

Therefore, between the two extremes outlined above, processor designers must aim for an optimum design of internal processor data paths, appropriate for the given instruction set and the targeted processor performance. This point will be discussed further in Section 12.6, when we discuss a technique known as *operand forwarding*.

As mentioned above, the important question of defining *program* (or *thread*) *state* and *processor state* must also be addressed. If a context switch, interrupt or exception occurs, the program/thread state and processor state must be saved, and then restored at a later time when the same program/thread resumes. From the programmer's point of view, the state should correspond to a point in the machine language program at which the previous instruction has completed execution, but the next one has not started.

In a multiple-issue processor, clearly this requires careful thought—since, at any time, as many as a couple of dozen instructions may be in various stages of execution.

A processor of the type described here is often designed with hardware support for *multi-threading*, which requires maintaining thread status of multiple threads, and switching between threads; this type of design is discussed further in Section 12.12.

Note also that, in Fig. 12.4 and Fig. 12.5, we have separated control elements from data flow elements and functional units in the processor—and in fact shown only the latter. Design of the control logic needed for the processor will not be discussed in this chapter in any degree of detail, beyond the brief overview contained in Note 12.2.

Note 12.2

The processor designer must select the architectural components to be included in the processor—for example a *reorder buffer* of a particular type, a specific method of *operand forwarding*, a specific method of *branch prediction*, and so on. The designer must also specify fully the algorithms which will govern the working of the selected architectural components. These algorithms are very similar to the algorithms we write in higher level programming languages, and are written using similar languages. These algorithms specify the control logic that would be needed for the processor, which would be finally realized in the form of appropriate digital logic circuits.

Given the complexity of modern systems, the task of translating algorithmic descriptions of processor functions into digital logic circuits can only be carried out using very sophisticated VLSI design software. Such software offers a wide range of functionality; *simulation* software is used to verify the correctness of the selected algorithm; *logical design* software translates the algorithm into a digital circuit; *physical design* software translates the logical circuit design into a physical circuit which can be built using VLSI, while *design verification* software verifies that the physical design does not violate any constraints of the underlying circuit fabrication technology.

All the architectural elements and control logic which is being described in this chapter can thus be translated into a physical design and then realized in VLSI. This is how processors and other digital systems are designed and built today. For our purposes in this chapter, however, it is not necessary to go into the details of how the required circuits and control logic are to be realized in VLSI.

We take the view that the architect decides *what* is to be designed, and then the circuit designer designs and realizes the circuit accordingly. In other words, our subject matter is restricted to the functions of the architect, and does not extend to circuit design—i.e. to the question of *how* a particular function is to be realized in VLSI. We assume that any required control logic which can be clearly specified can be implemented.

12.5 COMPILER-DETECTED INSTRUCTION LEVEL PARALLELISM

In the process of translating a sequential source program into machine language, the compiler performs extensive syntactic and semantic analysis of the source program. Therefore computer scientists have considered carefully the question of whether the compiler can uncover the instruction level parallelism which is implicit in the program. As we shall see, there are several ways in which the compiler can contribute to the exploitation of implicit instruction level parallelism.

One relatively simple technique which the compiler can employ is known as *loop unrolling*, by which independent instructions from multiple successive iterations of a loop can be made to execute in parallel.

Unrolling means that the body of the loop is repeated n times for n successive values of the control variable—so that one iteration of the transformed loop performs the work of n iterations of the original loop.

Example 12.4 Loop unrolling

Consider the following body of a loop in a user program, where all the variables except the loop control variable i are assumed to be floating point:

```
for i = 0 to 58 do
    c[i] = a[i]*b[i] - p*d[i];
```

Now suppose that machine code is generated by the compiler as though the original program had been written as:

```
for j = 0 to 52 step 4 do
{
  c[j] = a[j]*b[j] - p*d[j];
  c[j+1] = a[j+1]*b[j+1] - p*d[j+1];
  c[j+2] = a[j+2]*b[j+2] - p*d[j+2];
  c[j+3] = a[j+3]*b[j+3] - p*d[j+3];
}
c[56] = a[56]*b[56] - p*d[56];
c[57] = a[57]*b[57] - p*d[57];
c[58] = a[58]*b[58] - p*d[58];
```

Note carefully the values of loop variable `j` in the transformed loop.

The reader may verify, without too much difficulty, that the two program fragments are equivalent, in the sense that they perform the same computation. Of course the compiler does not transform one source program into another—it simply produces machine code corresponding to the second version, with the *unrolled* loop.

In the unrolled program fragment, the loop contains four independent instances of the original loop body—indeed this is the meaning of *loop unrolling*. Suppose machine code corresponding to the second program fragment is executing on a processor. Then clearly—if the processor has sufficient floating point arithmetic resources—instructions from the four loop iterations can be in progress in parallel on the various functional units.

It is clear that code length of the machine language program increases as a result of loop unrolling; this increase may have an effect on the cache hit ratio. Also, more registers are needed to exploit the instruction level parallelism within the longer unrolled loop. In such cases, techniques such as *register renaming*—discussed in Section 12.8—can allow greater exploitation of instruction level parallelism in the unrolled loop.

To discover and exploit the parallelism implicit in loops, as seen in Example 12.4, the compiler must perform the *loop unrolling* transformation to generate the machine code. Clearly, this strategy makes sense only if sufficient hardware resources are provided within the processor for executing instructions in parallel.

In the simple example above, the loop control variable in the original program goes from 0 to 58—i.e. its initial and final values are both known at compile time. If, on the other hand, the loop control values are not known at compile time, the compiler must generate code to calculate at run-time the control values for the unrolled loop.

Note that loop unrolling by the compiler does not in itself involve the detection of instruction level parallelism. But loop unrolling makes it possible for the compiler or the processor hardware to exploit a greater degree of instruction level parallelism. In Example 12.4, since the basic block making up the loop body becomes longer, it becomes possible for the compiler or processor to find a greater degree of parallelism amongst the instructions across the unrolled loop iterations.

Can the compiler also do the additional work of actually scheduling machine instructions on the hardware resources available on the processor? Or must this scheduling be necessarily performed *on the fly* by the processor control logic?

When the compiler schedules machine instructions for execution on the processor, the form of scheduling is known as *static scheduling*. As against this, instruction scheduling carried out by the processor hardware *on the fly* is known as *dynamic scheduling*, which has been introduced in Chapter 6 and will be discussed further later in this chapter.

If the compiler is to schedule machine instructions, then it must perform the required dependence analysis amongst instructions. This is certainly possible, since the compiler has access to full semantic information obtained from the original source program.

Example 12.5 Dependence across loop iterations

Consider the following loop in a source program, which appears similar to the loop seen in the previous example, but has a crucial new dependence built into it:

```
for i = 0 to 58 do
    c[i] = a[i]*b[i] - p*c[i-1];
```

Now the value calculated in the i^{th} iteration of the loop makes use of the value `c[i-1]` calculated in the previous iteration. This does not mean that the modified loop cannot be unrolled, but only that extra care should be taken to account for the dependence.

Dependences amongst references to simple variables, or amongst array elements whose index values are known at compile time (as in the two examples seen above), can be analyzed relatively easily at compile time.

But when pointers are used to refer to locations in memory, or when array index values are known only at run-time, then clearly dependence analysis is not possible at compile time. Therefore processor hardware must provide support at run-time for *alias analysis*—i.e. based on the respective effective addresses, to determine whether two memory accesses for read or write operations refer to the same location.

There is another reason why static scheduling by the compiler must be backed up by dynamic scheduling by the processor hardware. Cache misses, I/O interrupts, and hardware exceptions cannot be predicted

at compile time. Therefore, apart from *alias analysis*, the disruptions caused by such events in statically scheduled running code must also be handled by the dynamic scheduling hardware in the processor.

These arguments bring out a basic point—compiler detected instruction level parallelism also requires dynamic scheduling support within the processor. The fact that compiler performs extra work does not really make the processor hardware much simpler[4].

A further step in the direction of compiler detected instruction level parallelism and static scheduling can be the following:

Suppose each machine instruction specifies multiple operations—to be carried out in parallel within the processor, on multiple functional units. The machine language program produced by the compiler then consists of such multi-operation instructions, and their scheduling takes into account all the dependences amongst instructions.

Recall that conventional machine instructions specify one operation each—e.g. *load*, *add*, *multiply*, and so on. As opposed to this, multi-operation instructions would require a larger number of bits to encode. Therefore processors with this type of instruction word are said to have *very long instruction word* (VLIW). A preliminary discussion of this concept has been included in Chapter 4 of the book.

A little further refinement of this concept brings us to the so-called *explicitly parallel instruction computer* (EPIC). The EPIC instruction format can be more flexible than the fixed format of multi-operation VLIW instruction; for example, it may allow the compiler to encode explicitly dependences between operations.

Another possibility is that of having *predicated instructions* in the instruction set, whereby an instruction is executed only if the hardware condition (predicate) specified with it holds true. Such instructions would result in reduced number of conditional branch instructions in the program, and could thereby lower the number of pipeline flushes.

The aim behind VLIW and EPIC processor architecture is to assign to the compiler primary responsibility for the parallel exploitation of plentiful hardware resources of the processor. In theory, this would simplify the processor hardware, allowing for increased aggregate processor throughput. Thus this approach would, in theory, provide a third alternative to the RISC and CISC styles of processor architecture.

In general, however, it is fair to say that VLIW and EPIC concepts have not fulfilled their original promise. Intel Itanium 64-bit processors make up the most well-known processor family of this class. Experience with that processor showed, as was argued briefly above, that processor hardware does not really become simpler even when the compiler bears primary responsibility for the detection and exploitation of instruction level parallelism. Events such as interrupts and cache misses remain unpredictable, and therefore execution of operations at run-time cannot follow completely the static scheduling specified in VLIW/ EPIC instructions by the compiler; dynamic scheduling is still needed.

Another practical difficulty with compiler detected instruction level parallelism is that the source program may have to be recompiled for a different processor model of the same processor family. The reason is simple: such a compiler depends not only on the instruction set architecture (ISA) of the processor family, but also on the hardware resources provided on the specific processor model for which it generates code.

---

[4] Recall in this context the basic argument for RISC architecture, whereby the instruction set is *reduced* for the sake of higher processor throughput. A similar trade-off between hardware and software complexity does not exist when the compiler performs static scheduling of instructions on a superscalar processor.

For highly compute-intensive applications which run on dedicated hardware platforms, this strategy may well be feasible and it may yield significant performance benefits. Such special-purpose applications are fine-tuned for a given hardware platform, and then run for long periods on the same dedicated platform.

But commonly used programs such as word processors, web browsers, and spreadsheets must run without recompilation on all the processors of a family. Most users of software do not have source programs to recompile, and all the processors of a family are expected to be instruction set compatible with one another. Therefore the role of compiler-detected instruction level parallelism is limited in the case of widely used general purpose application programs of the type mentioned.

12.6 OPERAND FORWARDING

We know that a superscalar processor offers opportunities for the detection and exploitation of instruction level parallelism—i.e. potential parallelism which is present within a single instruction stream. Exploitation of such parallelism is enhanced by providing multiple functional units and by other techniques that we shall study. True data dependences between instructions must of course be respected, since they reflect program logic. On the other hand, two independent instructions can be executed in parallel—or even out of sequence—if that results in better utilization of processor clock cycles.

We now know that pipeline *flushes* caused by conditional branch, indirect jump, and procedure return instructions lead to degradation in performance, and therefore attempts must be made to minimize them; similarly pipeline *stalls* caused by data dependences and cache misses also have adverse impact on processor performance.

Therefore the strategy should be to minimize the number of pipeline stalls and flushes encountered while executing an instruction stream. In other words, we must minimize wasted processor clock cycles within the pipeline and also, if possible, within the various functional units of the processor.

In this section, we take a look at a basic technique known as *operand forwarding*, which helps in reducing the impact of true data dependences in the instruction stream. Consider the following simple sequence of two instructions in a running program:

```
ADD       R1, R2, R3
SHIFTR    #4, R3, R4
```

The result of the ADD instruction is stored in destination register R3, and then shifted right by four bits in the second instruction, with the shifted value being placed in R4. Thus, there is a simple RAW *dependence* between the two instructions—the output of the first is required as input operand of the second.

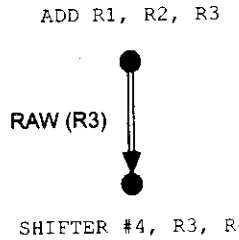


Fig. 12.6 RAW dependence between two instructions

In terms of our notation, this RAW dependence appears as shown in Fig. 12.6, in the form of a graph with two nodes and one edge.

In a pipelined processor, ideally the second instruction should be executed one stage—and therefore one clock cycle—behind the first. However, the difficulty here is that it takes one clock cycle to transfer ALU output to destination register R3, and then another clock cycle to transfer the contents of

register R3 to ALU input for the right shift. Thus a total of two clock cycles are needed to bring the result of the first instruction where it is needed for the second instruction. Therefore, as things stand, the second instruction above cannot be executed just one clock cycle behind the first.

This sequence of data transfers has been illustrated in Fig. 12.7 (a). In clock cycle T_k, ALU output is transferred to R3 over an internal data path. In the next clock cycle T_{k+1}, the content of R3 is transferred to ALU input for the right shift. When carried out in this order, clearly the two data transfer operations take two clock cycles.

But note that the required two transfers of data can be achieved in only one clock cycle if ALU output is sent to both R3 and ALU input in the same clock cycle—as illustrated in Fig. 12.7 (b). In general, if X is to be copied to Y, and in the next clock cycle Y is to be copied to Z, then we can just as well copy X to both Y and Z in one clock cycle.

If this is done in the above sequence of instructions, the second instruction can be just one clock cycle behind the first, which is a basic requirement of an instruction pipeline.

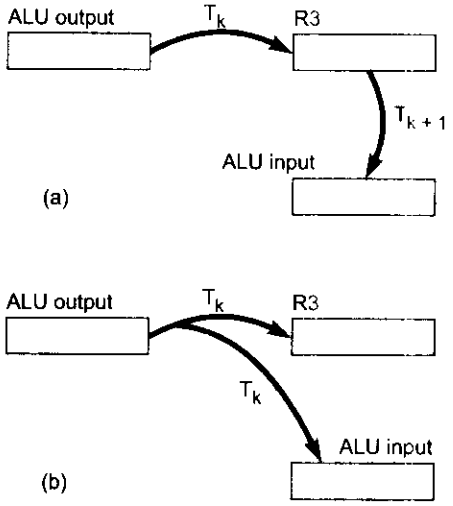


Fig. 12.7 Two data transfers (a) in sequence and (b) in parallel

In technical terms, this type of an operation within a processor is known as *operand forwarding*. Basically this means that, instead of performing two or more data transfers from a common source one after the other, we perform them in parallel. This can be seen as parallelism at the level of elementary data transfer operations within the processor. To achieve this aim, the processor hardware must be designed to detect and exploit *on the fly* all such opportunities for saving clock cycles. We shall see later in this chapter one simple and elegant technique for achieving this aim.

The benefits of such a technique are easy to see. The wait within a functional unit for its operand becomes shorter because, as soon as it is available, the operand is sent in one clock cycle, over the common data bus, to every destination where it is needed. We saw in the above example that thereby the common data bus remained occupied for one clock cycle rather than two clock cycles. Since this bus itself is a key hardware resource, its better utilization in this way certainly contributes to better processor performance.

The above reasoning applies even if there is an intervening instruction between ADD and SHIFTR. Consider the following sequence of instructions:

```
ADD     R1, R2, R3
SUB     R5, R6, R7
SHIFTR  #4, R3, R4
```

SHIFTR must be executed after ADD, in view of the RAW dependence. But there is no such dependence between SUB and any of the other two instructions, which means that SUB can be executed in program order, or before ADD, or after SHIFTR.

If SUB is executed in program order, then even without operand forwarding between ADD and SHIFTR, no processor clock cycle is lost, since SHIFTR does not directly follow ADD. But now suppose SUB is executed either before ADD, or after SHIFTR. In both these cases, SHIFTR directly follows ADD, and therefore operand forwarding proves useful in saving a processor cycle, as we have seen above.

Figure 12.8 shows the dependence graph of these three instructions. Since there is only one dependence in this instance amongst the three instructions, the graph in the figure has three nodes and only one edge.

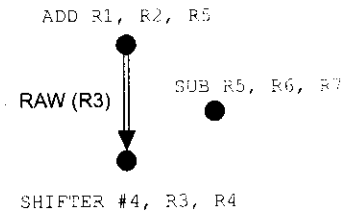


Fig. 12.8 Dependence graph of three instructions

But *why* should SUB be executed in any order other than program order?

The answer can only be this: to achieve better utilization of processor clock cycles. For example, if for some reason ADD cannot be executed in a given clock cycle, then the processor may well decide to execute SUB before it.

Therefore the processor must make *on the fly* decisions such as

(i) transferring ALU output in parallel to both R3 and ALU input, and/or
(ii) out of order execution of the instruction SUB.

This implies that the control logic of the processor must detect any such possibilities and generate the required control signals. This is in fact what is needed to implement *dynamic scheduling* of machine instructions within the processor.

Of course, to achieve performance speed-up through dynamic scheduling, considerable complexity must be added to processor control logic—but that is a price which must be paid for exploiting instruction level parallelism in the sequence of executing instructions; the complexity in achieving superscalar performance would of course be greater.

Machine instructions of a typical processor can be classified into *data transfer* instructions, *arithmetic and logic* instructions, *comparison* instructions, *transfer of control* instructions, and other miscellaneous instructions.

Of these, only the second group of instructions—i.e. arithmetic and logic instructions—actually alter the values of their operands. The other groups of instructions involve only transfers of data within the processor, between the processor and main memory, or between the processor and an I/O adapter.

Arithmetic and logic instructions are basically functions to be computed—either unary functions of the form $y = f(x)$, or binary functions of the form $y = f(x_1, x_2)$. And these computations are carried out by functional units such as *arithmetic and logic unit* (ALU), *floating point unit* (FPU), and so on. But even to get any computations done by these functional units, we need (i) transfer of operands to the inputs of functional units, and (ii) transfer of results back to registers or to the reorder buffer.

From the above arguments, it should be clear that data transfers make up a large proportion of the work of any processor. The need to fully utilize available hardware resources forces designers to pay close attention to the data transfers required not only for a single executing instruction, but also across multiple instructions. In this context, operand forwarding can be seen as a potentially powerful technique to reduce the number of clock cycles spent in carrying out the required data transfers within the processor.

In Fig. 12.4 and Fig. 12.5, we have not shown details of the data paths connecting the various elements within the processor. This is intentional, because the nature and number of data paths, their widths, their access mechanisms, *et cetera*, must be designed to be consistent with (i) the various hardware resources provided within the processor, and (ii) the target performance of the processor. Details of the data paths cannot be pinned down at an early stage, when the rest of the design is not yet completed.

We have discussed earlier a basic point related to any system performance: *there should be no performance bottlenecks in the system*. Clearly therefore the system of data paths provided within the processor should also not become a performance limiting element. A multiple issue processor targets $k > 1$ instruction issues per processor clock cycle. Hence the demands made on each of the elements of the processor—including cache memories, functional units, and internal data paths—would be k times greater.

REORDER BUFFER

The *reorder buffer* as a processor element was introduced and discussed briefly in Section 12.4. Since instructions execute in parallel on multiple functional units, the reorder buffer serves the function of bringing completed instructions back into an order which is consistent with program order. Note that instructions may *complete* in an order which is not related to program order, but must be *committed* in program order.

At any time, *program state* and *processor state* are defined in terms of instructions which have been committed—i.e. their results are reflected in appropriate registers and/or memory locations. The concepts of program state and processor state are important in supporting context switches and in providing precise exceptions.

Entries in the reorder buffer are completed instructions, which are queued in program order. However, since instructions do not necessarily complete in program order, we also need a flag with each reorder buffer entry to indicate whether the instruction in that position has completed.

Figure 12.9 shows a reorder buffer of size eight. Four fields are shown with each entry in the reorder buffer—instruction identifier, value computed, program-specified destination of the value computed, and a flag indicating whether the instruction has completed (i.e. the computed value is available).

In Fig. 12.9, the head of queue of instructions is shown at the top, arbitrarily labeled as instr[i]. This is the instruction which would be committed next—if it has completed execution. When this instruction commits,

its result value is copied to its destination, and the instruction is then removed from the reorder buffer. The next instruction to be issued in the *issue* stage of the instruction pipeline then joins the reorder buffer at its tail.

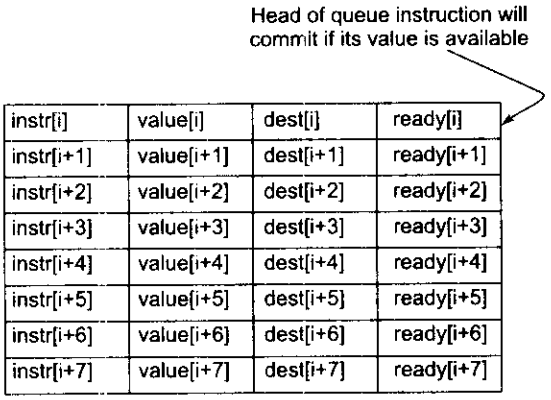


Fig. 12.9 Entries in a reorder buffer of size eight

If the instruction at the head of the queue has not completed, and the reorder buffer is full, then further issue of instructions is held up—i.e. the pipeline stalls—because there is no free space in the reorder buffer for one more entry.

The result value of any other instruction lower down in the reorder buffer, say value[i+k], can also be used as an input operand for a subsequent operation—provided of course that the instruction has completed and therefore its result value is available, as indicated by the corresponding flag ready[i+k]. In this sense, we see that the technique of *operand forwarding* can be combined with the concept of the reorder buffer.

It should be noted here that operands at the input latches of functional units, as well as values stored in the reorder buffer on behalf of completed but uncommitted instructions, are simply 'work in progress'. These values are not reflected in the state of the program or the processor, as needed for a context switch or for exception handling.

We now take a brief look at how the use of reorder buffer addresses the various types of dependences in the program.

(i) Data Dependences A RAW dependence—i.e. true data dependence—will hold up the execution of the dependent instruction if the result value required as its input operand is not available. As suggested above, operand forwarding can be added to this scheme to speed up the supply of the needed input operand as soon as its value has been computed.

WAR and WAW dependences—i.e. anti-dependence and output dependence, respectively—also hold up the execution of the dependent instruction and create a possible pipeline stall. We shall see below that the technique of *register renaming* is needed to avoid the adverse impact of these two types of dependences.

(ii) Control Dependences Suppose the instruction(s) in the reorder buffer belong to a branch in the program which should not have been taken—i.e. there has been a mis-predicted branch. Clearly then the

reorder buffer should be flushed along with other elements of the pipeline. Therefore the performance impact of control dependences in the running program is determined by the accuracy of branch prediction technique employed. The reorder buffer plays no direct role in the handling of control dependences.

(iii) Resource Dependences If an instruction needs a functional unit to execute, but the unit is not free, then the instruction must wait for the unit to become free—clearly no technique in the world can change that. In such cases, the processor designer can aim to achieve at least this: if a subsequent instruction needs to use another functional unit which is free, then the subsequent instruction can be executed out of order.

However, the reorder buffer queues and commits instructions in program order. In this sense, therefore, the technique of using a reorder buffer does not address explicitly the resource dependences existing within the instruction stream; with multiple functional units, the processor can still achieve out of order completion of instructions.

In essence, the conceptually simple technique of reorder buffer ensures that if instructions as programmed can be carried out in parallel—i.e. if there are no dependences amongst them—then they are carried out in parallel. But nothing clever is attempted in this technique to resolve dependences. Instruction issue and commit are in program order; program state and processor state are correctly preserved.

We shall now discuss a clever technique which alleviates the adverse performance effect of WAR and WAW dependences amongst instructions.

12.8 REGISTER RENAMING

Traditional compilers allocate registers to program variables in such a way as to reduce the main memory accesses required in the running program. In programming language C, in fact, the programmer can even pass a hint to the compiler that a variable be maintained in a processor register.

Traditional compilers and assembly language programmers work with a fairly small number of programmable registers. The number of programmable registers provided on a processor is determined by either

(i) the need to maintain backward instruction compatibility with other members of the processor family, or

(ii) the need to achieve reasonably compact instruction encoding in binary. With sixteen programmable registers, for example, four bits are needed for each register specified in a machine instruction.

Amongst the instructions in various stages of execution within the processor, there would be occurrences of RAW, WAR and WAW dependences on programmable registers. As we have seen, RAW is true data dependence—since a value written by one instruction is used as an input operand by another. But a WAR or WAW dependence can be avoided if we have more registers to work with. We can simply remove such a dependence by getting the two instructions in question to use two different registers.

But we must also assume that the *instruction set architecture* (ISA) of the processor is fixed—i.e. we cannot change it to allow access to a larger number of programmable registers. Rather, our aim here is to explore techniques to detect and exploit instruction level parallelism using a given instruction set architecture.

Therefore the only way to make a larger number of registers available to instructions under execution within the processor is to make the additional registers *invisible* to machine language instructions. Instructions *under execution* would use these additional registers, even if instructions making up the machine language program stored in memory cannot refer to them.

Let us suppose that we have several such additional registers available, to which machine instructions of the running program cannot make any direct reference. Of course these machine instructions do refer to programmable registers in the processor—and thereby create the WAR and WAW dependences which we are now trying to remove.

For example, let us say that the instruction:

```
FADD    R1, R2, R5
```

is followed by the instruction:

```
FSUB    R3, R4, R5
```

Both these instructions are writing to register R5, creating thereby a WAW dependence—i.e. output dependence—on register R5. Clearly, any subsequent instruction should read the value written into R5 by FSUB, and not the value written by FADD. Figure 12.10 shows this dependence in graphical notation.

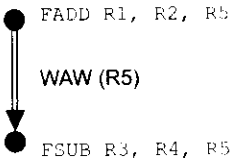


Fig. 12.10 WAW dependence

With additional registers available for use as these instructions execute, we have a simple technique to remove this output dependence.

Let FSUB write its output value to a register other than R5, and let us call that other register X. Then the instructions which use the value generated by FSUB will refer to X, while the instructions which use the value generated by FADD will continue to refer to R5. Now, since FADD and FSUB are writing to two different registers, the output dependence or WAW between them has been removed![5]

When FSUB *commits*, then the value in R5 should be updated by the value in X—i.e. the value computed by FSUB. Then the physical register X, which is not a program visible register, can be freed up for use in another such situation.

Note that here we have *mapped*—or *renamed*—R5 to X, for the purpose of storing the result of FSUB, and thereby removed the WAW dependence from the instruction stream. A pipeline stall will now not be created due to the WAW dependence.

In general, let us assume that instruction I_i writes a value into register R_k. At the time of instruction issue, we map this programmable register R_k onto a program invisible register X_m, so that when instruction I_j executes, the result is written into X_m rather than R_k. In this program invisible register X_m, the result value is available to any other instruction which is truly data dependent on I_j—i.e. which has RAW dependence on I_j.

If any instruction other than I_j is also writing into R_k, then that instance of R_k will be mapped into some other program invisible register X_n. This renaming resolves the WAW dependence between the two instructions involving R_k. When instruction I_j commits, the value in X_m is copied back into R_k, and the program invisible register X_m is freed up for reuse by another instruction.

[5] In fact the processor may also rename R5 in FADD to another program invisible register, say Y. But clearly the argument made here still remains valid.

A similar argument applies if I_j is reading the value in R_k, and a subsequent instruction is writing into R_k—i.e. there is a WAR dependence between them.

The technique outlined, which can resolve WAR and WAW dependences, is known as *register renaming*. Both these dependences are caused by a subsequent instruction writing into a register being used by a previous instruction. Such dependences do not reflect program logic, but rather the use of a limited number of registers.

Let us now consider a simple example of WAR dependence, i.e. of anti-dependence. The case of WAW dependence would be very similar.

Example 12.6 Register renaming and WAR dependence

Assume that the instructions:

```
FADD    R6, R7, R2
FADD    R2, R3, R5
```

are followed later in the program by the instruction:

```
FSUB    R1, R3, R2
```

The first FADD instruction is writing a value into R2, which the second FADD instruction is using, i.e. there is true data dependence between these two instructions. Let us assume that, when the first FADD instruction executes, R2 is mapped into program invisible register X_m.

The latter FSUB instruction is writing another value into R2. Clearly, the second FADD (and other intervening instructions before FSUB) should see the value in R2 which is written by the first FADD—and not the value written by FSUB. Figure 12.11 shows these two dependences in graphical notation.

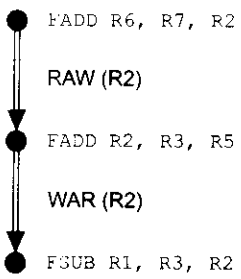


Fig. 12.11 RAW and WAR dependences

With register renaming, it is a simple matter to resolve the WAR anti-dependence between the second FADD and FSUB.

As mentioned, let X_m be the program invisible register to which R2 has been mapped when the first FADD executes. This is then the remapped register to which the second FADD refers for its first data operand.

Let FSUB write its output to a program invisible register other than X_m, which we denote by X_n. Instructions which use the value written by FSUB refer to X_n, while instructions which use the value written by the first FADD refer to X_m.

The WAR dependence between the second FADD and FSUB is removed; but the RAW dependence between the two FADD instructions is respected via X_m.

When the first FADD commits, the value in X_m is transferred to R2 and program invisible register X_m is freed up; likewise, later when FSUB commits. the value in X_n is transferred to R2 and program invisible register X_m is freed up.

Thus we see that register renaming removes WAR and WAW dependences from the instruction stream by re-mapping programmable registers to a larger pool of program invisible registers. For this, the processor must have extra registers to handle instructions under execution, but these registers do not appear in the instruction set.

Consider true data dependence, i.e. RAW dependence, between two instructions. Under register renaming, the write operation and the subsequent read operation both occur on the same program invisible register. Thus RAW dependence remains intact in the instruction stream—as it should, since it is true data dependence. As seen above, its impact on the pipeline operation can be reduced by operand forwarding.

Dependences are also caused by reads and writes to memory locations. In general, however, whether two instructions refer to the same memory location can only be known after the two effective addresses are calculated during execution. For example, the two memory references 2000[R1] and 4000[R3] occurring in a running program may or may not refer to the same memory location—this cannot be resolved at compile time.

Resolution of whether two memory references point to the same memory location is known as *alias analysis*, which must be carried out on the basis of the two effective memory addresses. If a *load* and a *store* operation to memory refer to two different addresses, their order may be interchanged. Such capability can be built into the load-store unit—which in essence operates as another functional unit of the processor.

An elegant implementation of register renaming and operand forwarding in a high performance processor was seen as early as in 1967—even before the term *register renaming* was coined. This technique—which has since become well-known as *Tomasulo's algorithm*—is described in the next section.

12.9 TOMASULO'S ALGORITHM

In the IBM 360 family of computer systems of 1960s and 1970s, model 360/91 was developed as a high performance system for scientific and engineering applications, which involve intensive floating point computations. The processor in this system was designed with multiple floating point units, and it made use of an innovative algorithm for the efficient use of these units. The algorithm was based on operand forwarding over a common data bus, with tags to identify sources of data values sent over the bus.

The algorithm has since become known as *Tomasulo's algorithm*, after the name of its chief designer[6]; what we now understand as *register renaming* was also an implicit part of the original algorithm.

Recall that, for register renaming, we need a set of program invisible registers to which programmable registers are re-mapped. Tomasulo's algorithm requires these program invisible registers to be provided with reservation stations of functional units.

Let us assume that the functional units are internally pipelined, and can complete one operation in every clock cycle. Therefore each functional unit can initiate one operation in every clock cycle—provided of course that a reservation station of the unit is ready with the required input operand value or values. Note that the exact depth of this functional unit pipeline does not concern us for the present.

[6] See *An efficient algorithm for exploiting multiple arithmetic units*, by R. M. Tomasulo, IBM Journal of Research & Development 11:1, January 1967. A preliminary discussion on Tomasulo's algorithm was included in Chapter 6.

Figure 12.12 shows such a functional unit connected to the common data bus, with three reservation stations provided on it.

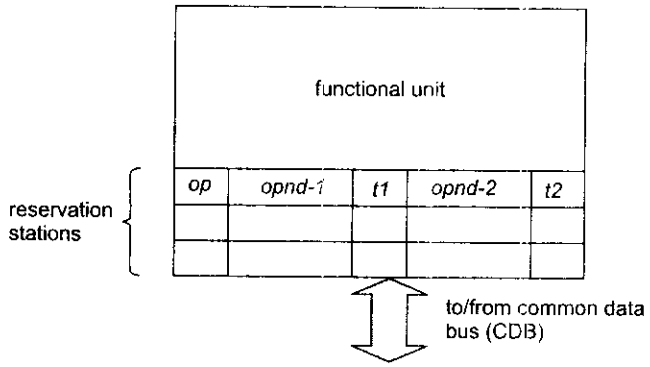


Fig. 12.12 Reservation stations provided with a functional unit

The various fields making up a typical reservation station are as follows:

op	operation to be carried out by the functional unit
opnd-1 & *opnd-2*	two operand values needed for the operation
t1 & *t2*	two source tags associated with the operands

When the needed operand value or values are available in a reservation station, the functional unit can initiate the required operation in the next clock cycle.

At the time of instruction issue, the reservation station is filled out with the operation code (*op*). If an operand value is available, for example in a programmable register, it is transferred to the corresponding source operand field in the reservation station.

However, if the operand value is not available at the time of issue, the corresponding source tag (*t1* and/or *t2*) is copied into the reservation station. The source tag identifies the source of the required operand. As soon as the required operand value is available at its source—which would be typically the output of a functional unit—the data value is forwarded over the common data bus, along with the source tag. This value is copied into all the reservation station operand slots which have the matching tag.

Thus operand forwarding is achieved here with the use of tags. All the destinations which require a data value receive it in the same clock cycle over the common data bus, by matching their stored operand tags with the source tag sent out over the bus.

Example 12.7 Tomasulo's algorithm and RAW dependence

Assume that instruction I1 is to write its result into R4, and that two subsequent instructions I2 and I3 are to read—i.e. make use of—that result value. Thus instructions I2 and I3 are truly data dependent (RAW dependent) on instruction I1. See Fig. 12.13.

Assume that the value in R4 is not available when I2 and I3 are issued; the reason could be, for example, that one of the operands needed for I1 is itself not available. Thus we assume that I1 has not even started executing when I2 and I3 are issued.

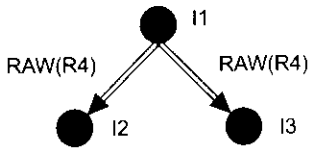


Fig. 12.13 Example of RAW dependences

When I2 and I3 are issued, they are parked in the reservation stations of the appropriate functional units. Since the required result value from I1 is not available, these reservation station entries of I2 and I3 get source tag corresponding to the output of I1—i.e. output of the functional unit which is performing the operation of I1.

When the result of I1 becomes available at its functional unit, it is sent over the common data bus along with the tag value of its source—i.e. output of functional unit.

At this point, programmable register R4 as well as the reservation stations assigned to I2 and I3 have the matching source tag—since they are waiting for the same result value, which is being computed by I1.

When the tag sent over the common data bus matches the tag in any destination, the data value on the bus is copied from the bus into the destination. The copy occurs at the same time into all the destinations which require that data value. Thus R4 as well as the two reservation stations holding I2 and I3 receive the required data value, which has been computed by I1, at the same time over the common data bus.

Thus, through the use of source tags and the common data bus, in one clock cycle, three destination registers receive the value produced by I1—programmable register R4, and the operand registers in the reservation stations assigned to I2 and I3

Let us assume that, at this point, the second operands of I2 and I3 are already available within their corresponding reservation stations. Then the operations corresponding to I2 and I3 can begin in parallel as soon as the result of I1 becomes available—since we have assumed here that I2 and I3 execute on two separate functional units.

It may be noted from Example 12.7 that, in effect, programmable registers become renamed to operand registers within reservation stations, which are program invisible. As we have seen in the previous section, such renaming also resolves anti-dependences and output dependences, since the target register of the dependent instruction is renamed in these cases to a different program invisible register.

Example 12.8 Combination of RAW and WAR dependence

Let us now consider a combination of RAW and WAR dependences.

Assume that instruction I1 is to write its result into R4, a subsequent instructions I2 is to read that result value, and a latter subsequent instruction I3 is then to write *its* result into R4. Thus instruction I2 is truly data dependent (RAW dependent) on instruction I1, but I3 is anti-dependent (WAR dependent) on I2. See Fig. 12.14.

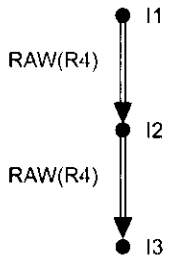


Fig. 12.14 Example of RAW & WAR dependences

As in the previous example, and keeping in mind similar possibilities, let us assume once again that the output of I1 is not available when I2 and I3 are issued, thus R4 has the source tag value corresponding to the output of I1.

When I2 is issued, it is parked in the reservation station of the appropriate functional unit. Since the required result value from I1 is not available, the reservation station entry of I2 also gets the source tag corresponding to the output of I1—i.e. the same source tag value which has been assigned to register R4, since they are both awaiting the same result.

The question now is: Can I3 be issued even before I1 completes and I2 starts execution?

The answer is that, with register renaming—carried out here using source tags—I3 can be issued even before I2 starts execution.

Recall that instruction I2 is RAW dependent on I1, and therefore it has the correct source tag for the output of I1. I2 will receive its required input operand as soon as that is available, when that value would also be copied into R4 over the common data bus. This is exactly what we observed in the previous example.

But suppose I3 is issued even before the output of I1 is available. Now R4 should receive the output of I3 rather than the output of I1. This is simply because, in register R4, the output of I1 is programmed to be overwritten by the output of I3.

Thus, when I3 is issued, R4 will receive the source tag value corresponding to the output of I3—i.e. the functional unit which performs the operation of I3. Its previous source tag value corresponding to the output of I1 will be overwritten.

When the output of I1 (finally) becomes available, it goes to the input of I2, but not to register R4, since this register's source tag now refers to I3. When the output of I3 becomes available, it goes correctly to R4 because of the matching source tag.

For simplicity of discussion, we have not tracked here the outputs of I2 and I3. But the student can verify easily that the two data transfers described above are consistent with the specified sequence of three instructions and the specified dependences.

Example 12.9 Scheduling across multiple iterations

Consider now the original iterative program loop discussed in Example 12.4.

Let us assume that, without any unrolling by the compiler, this loop executes on a processor which provides branch prediction and implements Tomasulo's algorithm. If instructions from successive loop iterations are available in the processor at one time—because of successful branch prediction(s)—and if floating point units are available, then instructions from successive iterations can execute at one time, in parallel.

But if instructions from multiple iterations are thus executing in parallel within the processor—at one time—then the net effect of these hardware techniques in the processor is the same as that of an unrolled loop. In other words, the processor hardware achieves *on the fly* what otherwise would require unrolling assistance from the compiler!

Even the dependence shown in Example 12.5 across successive loop iterations is handled in a natural way by branch prediction and Tomasulo's algorithm. Basically this dependence across loop iterations becomes RAW dependence between instructions, and is handled in a natural way by source tags and operand forwarding.

This example brings our clearly how a particular method of exploiting parallelism—*loop unrolling*, in this case—can be implemented either by the compiler or, equivalently, by clever hardware techniques employed within the processor.

Example 12.9 illustrates the combined power of sophisticated hardware techniques for dynamic scheduling and branch prediction. With such efficient techniques becoming possible in hardware, the importance of compiler-detected parallelism (Section 12.5) diminishes somewhat in comparison.

Example 12.10 Calculation of processor clock cycles

Let us consider the number of clock cycles it takes to execute the following sequence of machine instructions. We shall count clock cycles starting from the last clock cycle of instruction 1, so that the answer is independent of the depth of instruction pipeline.

```
1    LOAD     mem-a, R4
2    FSUB     R7, R4, R4
3    STORE    mem-a, R4
4    FADD     R4, R3, R7
5    STORE    mem-b, R7
```

We shall assume that (a) one instruction is issued per clock cycle, (b) floating point operations take two clock cycles each to execute, and (c) memory operations take one clock cycle each when there is L1 cache hit.

If we add the number of clock cycles needed for each instruction, we get the total as 1+2+1+2+1 = 7. However, if no operand forwarding is provided, the RAW dependences on registers R4 and R7 will cost three additional clock cycles (recall Fig. 12.7), for a total of 10 clock cycles for the given sequence of instructions.

With operand forwarding—which is built into Tomasulo's algorithm—one clock cycle is saved on account of each RAW dependence—i.e. between (i) instructions 1 and 2, (ii) instructions 2 and 3, and (ii) instructions 4 and 5.

Thus the total number of clock cycles required, counting from the last clock cycle of instruction 1, is 7. With the assumptions as made here, there is no further scope to schedule these instructions in parallel.

In Tomasulo's algorithm, use of the common data bus and operand forwarding based on source tags results in *decentralized control* of the multiple instructions in execution. In the 1960s and 1970s, Control Data Corporation developed supercomputers CDC 6600 and CDC 7600 with a *centralized* technique to exploit instruction level parallelism.

In these supercomputers, the processor had a centralized *scoreboard* which maintained the status of functional units and executing instructions (see Chapter 6). Based on this status, processor control logic governed the issue and execution of instructions. One part of the scoreboard maintained the status of every instruction under execution, while another part maintained the status of every functional unit. The scoreboard itself was updated at every clock cycle of the processor, as execution progressed.

12.10 BRANCH PREDICTION

The importance of branch prediction for multiple issue processor performance has already been discussed in Section 12.3. About 15% to 20% of instructions in a typical program are branch and jump instructions, including procedure returns. Therefore—if hardware resources are to be fully utilized in a superscalar processor—the processor must start working on instructions beyond a branch, even before the branch instruction itself has completed. This is only possible through some form of branch prediction.

What can be the logical basis for branch prediction? To understand this, we consider first the reasoning which is involved if one wishes to predict the result of a tossed coin.

Note 12.3 Predicting the outcome of a tossed coin

Can one predict the result of a single coin toss?

If we have prior knowledge—gained somehow—that the coin is unbiased, then the answer is a clear NO, in the sense that both possible outcomes *head* and *tail* are equally probable. The only possible prediction one can make in this case is that the coin will come up either *head* or *tail*—i.e. a prediction which is of no practical value!

But how can we come to have prior knowledge that a coin is unbiased? Logically, the only knowledge we can have about a coin is obtained through observations of outcomes in successive tosses. Therefore, the more realistic situation we must address is that we have no prior knowledge about the coin being either unbiased or biased. Having received a coin, any inference we make about it—i.e. whether it is biased or not—can only be on the basis of observations of outcomes of successive tosses of the coin.

In such a situation of no prior knowledge, assume that a coin comes up *head* in its first two tosses. Then simple conditional probability theory predicts that the third toss of the coin has a higher probability of coming up *head* than of coming up *tail*.

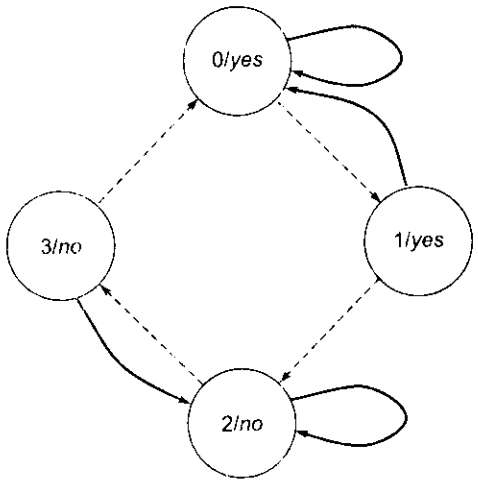
This is a straightforward example of Bayesian reasoning using conditional probabilities, named after Rev. Thomas Bayes [1702–1761]. French Mathematician Laplace [1749–1827] later addressed this and related questions and derived a formula to calculate the respective conditional probabilities[7].

[7] For a detailed discussion, with applications, the reader may refer to the book *Artificial Intelligence: A Modern Approach*, by Russell and Norvig, Pearson Education.

Like tossed coins, outcomes of conditional branches in computer programs also have *yes* and *no* answers— i.e. a branch is either taken or not taken. But outcomes of conditional branches are in fact biased—because there is strong correlation between (a) successive branches taken at the same conditional branch instruction in a program, and (b) branches taken at two different conditional branch instructions in the same program.

This is how programs behave, i.e. such correlation is an essential property of real-life programs. And such correlation provides the logical basis for branch prediction. The issue for processor designers is how to discover and utilize this correlation *on the fly* —without incurring prohibitive overhead in the process.

A basic branch prediction technique uses a so-called *two-bit predictor*. A two-bit counter is maintained for every conditional branch instruction in the program. The two-bit counter has four possible states; these four states and the possible transitions between these states are shown in Fig. 12.15.



In states 0 & 1: Branch taken
In states 2 & 3 Branch no taken

Solid line: Correct predication.
Broken line: incorrect prediction

Fig. 12.15 State transition diagram of 2-bit branch predictor[8]

When the counter state is 0 or 1, the respective branch is predicted as *taken*; when the counter state is 2 or 3, the branch is predicted as *not taken*. When the conditional branch instruction is executed and the actual branch outcome is known, the state of the respective two-bit counter is changed as shown in the figure using solid and broken line arrows.

When two successive predictions come out wrong, the prediction is changed from *branch taken* to *branch not taken*, and *vice versa*. In Fig. 12.14, state transitions made on mis-predictions are shown using broken line arrows, while solid line arrows show state transitions made on predictions which come out right.

[8] Note that Fig. 12.14 is a slightly redrawn version of the state transition diagram shown earlier in Fig. 6.19 (b).

This scheme uses a two-bit counter for every conditional branch, and there are many conditional branches in the program. Overall, therefore, this branch prediction logic needs a few kilobytes or more of fast memory. One possible organization for this branch prediction memory is in the form of an array which is indexed by low order bits of the instruction address. If twelve low order bits are used to define the array index, for example, then the number of entries in the array is 4096[9].

To be effective, branch prediction should be carried out as early as possible in the instruction pipeline. As soon as a conditional branch instruction is decoded, branch prediction logic should predict whether the branch is taken. Accordingly, the next instruction address should be taken either as the branch target address (i.e. branch is taken), or the sequentially next address in the program (i.e. branch is not taken).

Can branch prediction be carried out even before the instruction is decoded—i.e. at the instruction fetch stage? Yes, if a so-called *branch target buffer* is provided which has a history of recently executed conditional branches. The branch target buffer is organized as an associative memory accessed by the instruction address; this memory provides quick access to the prediction and the target instruction address needed.

In some programs, whether a conditional branch is taken or not taken correlates better with other conditional branches in the program—rather than with the earlier history of outcomes of the same conditional branch. Accordingly, *correlated predictors* can be designed, which generate a branch prediction based on whether other conditional branches in the program were taken or not taken.

Branch prediction based on the earlier history of the same branch is known as *local prediction*, while prediction based on the history of other branches in the program is known as *global prediction*. A *tournament predictor* uses (i) a global predictor, (ii) a local predictor, and (iii) a *selector* which selects one of the two predictors for prediction at a given branch instruction. The selector uses a two-bit counter per conditional branch—as in Fig.12.14—to choose between the global and local predictors for the branch. Two successive mis-predictions cause a switch from the local predictor to the global predictor, and *vice versa*; the aim is to infer which predictor works better for the particular branch.

The common element in all these cases is that branch prediction relies on the correlation detected between branches taken or not taken in the running program—and for this, an efficient hardware implementation of the required prediction logic is required.

Considerations outlined here apply also to *jump prediction*, which is applicable in indirect jumps, *computed go to* statements (used in FORTRAN), and *switch* statements (used in C and C++). Procedure returns can also benefit from a form of jump prediction. The reason is that the address associated with a procedure return is obtained from the runtime procedure stack in main memory; therefore a correct prediction of the return address can save memory access and a few processor clock cycles.

It is also possible to design the branch prediction logic to utilize information gleaned from a prior *execution profile* or *execution trace* of the program. If the same program is going to run on dedicated hardware for years—say for an application such as weather forecasting—then such special effort put into speeding up the program on that dedicated hardware can pay very good dividend over the life of the application. Suppose the execution trace informs us that a particular branch is taken 95% of the time, for example. Then it is a good idea to 'predict' the particular branch as always taken—in this case, we are assured that 95% of the predictions made will be correct!

[9] Clearly, if two conditional branch instructions happen to have the same low order bits, then their predictions will become 'intermingled'. But the probability of two ore more such instructions being in execution at the same time would be quite low.

As we discussed in Section 12.3, under any branch prediction scheme, a mis-predicted branch means that subsequent instructions must be flushed from the pipeline. It should of course be noted here that the actual result of a conditional branch instruction—as against its predicted result—is only known when the instruction completes execution.

Speculative Execution Instructions executed on the basis of a predicted branch, before the actual branch result is known, are said to involve *speculative execution*.

If a branch prediction turns out to be correct, the corresponding speculatively executed instructions must be committed. If the prediction turns out to be wrong, the effects of corresponding speculative operations carried out within the processor must be cleaned up, and instructions from another branch of the program must instead be executed.

As we have seen in Example 12.2, the strategy results in net performance gain if branch predictions are made with sufficiently high accuracy. The performance benefit of branch prediction can only be gained if prediction is followed by speculative execution.

A conventional processor fetches one instruction after another—i.e. it does not look ahead into the forthcoming instruction stream more than one instruction at a time. To support a deeper and wider—i.e. multiple issue—instruction pipeline, it is necessary for branch prediction and dynamic scheduling logic to look further out into the forthcoming instruction stream. In other words, more of the likely future instructions need to be examined in support of multiple issue scheduling and branch prediction.

Instruction window—or simply *window*—is the special memory provided upstream of the fetch unit in the processor to thus look ahead into the forthcoming instruction stream. For the targeted processor performance, the processor designers must integrate and balance the hardware techniques of branch prediction, dynamic scheduling, speculative execution, internal data paths, functional units, and an instruction window of appropriate size.

12.11 LIMITATIONS IN EXPLOITING INSTRUCTION LEVEL PARALLELISM

There is no such thing as free lunch!—an American proverb.

Technology is about trade-offs—and therefore it will come as no surprise to the student to learn that there are practical limits on the amount of instruction level parallelism which can be exploited in a single executing instruction stream. In this section we shall try to identify in broad terms some of the main limiting factors[10].

Consider as an example a multiple issue processor which targets four instruction issues per clock cycle, and has eight stages in the instruction pipeline. Clearly, in this processor at one time as many as thirty two instructions may be in different stages of fetch, decode, issue, execute, write result, commit, and so on—and each stage in the processor must handle four instructions in every clock cycle.

Assuming that 15% of the executing instructions are branches and jumps, the processor would handle at one time four to five such instructions—i.e. multiple predicted branches would be executing at one time.

[10] The interested student may read *Limits of Instruction-level Parallelism*, by D.W. Wall, Research Report 93/6, Western Research Laboratory, Digital Equipment Corporation, November 1993. Note 12.4 below is a brief summary of this technical report.

Similarly, multiple loads and stores would be in progress at one time. Also, dynamic scheduling would require a fairly large instruction window, to maintain the issue rate at the targeted four instructions per clock cycle.

Consider the instruction window. Instructions in the window must be checked for dependences, to support out of order issue. This requires associative memory and its control logic, which means an overhead in chip area and power consumption; such overhead would increase with window size. Similarly, any form of checking amongst executing instructions—e.g. checking addresses of main memory references, for alias analysis—would involve overhead which increases with issue multiplicity k. In turn, such increased overhead in aggressive pursuit of instruction level parallelism would adversely impact processor clock speed which is achievable, for a given VLSI technology.

Also, with greater issue multiplicity k, there would be higher probability of less than k instructions being issued in some clock cycles. The reason for this can be simply that a functional unit is not available, or that true RAW dependences amongst instructions hold up instruction issue. This would result in missing the target performance of the processor in actual applications, in terms of issue multiplicity k. Let us say processor A has $k = 6$ but it is only 60% utilized on average in actual applications; processor B, with the same instruction set but with $k = 4$, might have faster clock rate and also higher average utilization, thus giving better performance than A on actual applications.

The increased overhead also necessitates a larger number of stages in the instruction pipeline, so as to limit the total delay per stage and thereby achieve faster clock cycle; but a longer pipeline results in higher cost of flushing the pipeline. Thus the aggregate performance impact of increased overhead finally places limits on what is achievable in practice with aggressively superscalar, VLIW and EPIC architecture.[11]

Basically, the increased overhead required within the processor implies that:

(i) To support higher multiplicity of instruction issue, the amount of control logic required in the processor increases disproportionately, and

(ii) For higher throughput, the processor must also operate at a high clock rate.

But these two design goals are often at odds, for technical reasons of circuit design, and also because there are practical limits on the amount of power the chip can dissipate.

Power consumption of a chip is roughly proportional to $N \times f$, where N is the number of devices on the chip, and f is the clock rate. The number of devices on the chip is largely determined by the fabrication technology being used, and power consumption must be held within the limits of the heat dissipation possible.

Therefore the question for processor designers is: For a targeted processor performance, how best to select and utilize the various chip resources available, within the broad *design constraints* of the given circuit technology?

The student may recall that this was the introductory theme of this chapter (Section 12.1), and should note that such design trade-offs are shaping processor design today. To achieve their goals, processor designers make use of extensive software simulations of the processor, using various benchmark programs within the target range of applications. The designers' own experience and insights supplement the simulation results in the process of generating solutions to the actual problems of processor design.

[11] In this connection, see also the discussion in the latter part of Section 12.5.

Emergence of hardware support for multi-threading and of multi-core chips, which we shall discuss in the next section, is due in part to the practical limits which have been encountered in exploiting the implicit parallelism within a single instruction stream.

Note 12.4 Wall's Study on Instruction Level Parallelism

In this section, we have discussed the primary factor limiting the amount of instruction level parallelism which can be exploited in a sequence of executing instructions.

The report by D. W. Wall cited above was the result of a landmark empirical investigation into the amount of instruction level parallelism present in real-life programs. Any detailed discussion on the subject would benefit from a study of this report, and accordingly this note presents a brief summary of the report.

With reference to the overall design space which is available to the processor designer, Wall's report says:

Moreover, this space is multi-dimensional, because parallelism analysis consists of an ever-growing body of complementary techniques. The payoff of one choice depends strongly on its context in the other choices made. The purpose of this study is to explore that multi-dimensional space, and provide some insight about the importance of different techniques in different contexts.

The report was based on analysis of the instruction level parallelism present in 18 real-life programs; the number of combinations of processor features tried out during the analysis—i.e. 'points' in the processor design space—was more than 350.

Of the eighteen programs analyzed, twelve were from the standard SPEC92 benchmark suite, three were common utility programs, and three were engineering applications. The programs were compiled into machine language for the MIPS R3000 processor. Table 12.1 presents some information about these programs.

The technique employed in analyzing instruction level parallelism in these programs is known as *oracle-driven trace-based simulation*. For this, a complete trace of the program instructions executed is obtained from a previous run; the trace includes data addresses, results of branches and jumps, and so on. A scheduling algorithm is then used to pack these instructions, as tightly as possible, into a sequence of processor cycles.

As mentioned above, the simulation of processor performance was carried out at more than 350 different points in the space of possible processor configurations. These points differed from each other in the type of parallelism detection techniques used. At each such processor configuration point, an *oracle*[12] built into the simulator provided the scheduling decision, one by one, for each executed instruction. In other words, for a given processor configuration, the simulator had functionality built into it to determine the earliest possible schedule for each executed instruction of the program.

For each program, the final schedule of instructions generated by the simulator showed how the program would execute on a processor having the given configuration, as defined by the techniques used to exploit instruction level parallelism. The degree of parallelism obtained was then calculated from the simulation result.

[12] In the world of ancient Greece, an *oracle* was a power which could predict future events; one well-known and presumably reliable oracle was at the temple of Delphi.

For example, suppose one of the programs listed in Table 12.1 executed thirty million instructions, as seen from its execution trace. Suppose further that, for a given processor configuration, these instructions could be packed into six million processor cycles. Then the average degree of parallelism obtained for this program, for this particular processor configuration, would be 30/6 = 5.

Techniques Explored The range of techniques which was explored in the study to detect and exploit instruction level parallelism is summarized briefly below:

Register renaming—with (a) infinite number of registers as renaming targets, (b) finite number of registers, and (c) no register renaming.

Alias analysis—with (a) perfect alias analysis, (b) two intermediate levels of alias analysis, and (c) no alias analysis.

Branch prediction—with (a) perfect branch prediction, (b) three hardware-based branch prediction schemes, (c) three profile-based branch prediction schemes, and (d) no branch prediction. Hardware predictors used a combination of local and global tables, with different total table sizes. Some branch fanout limits were also applied.

Indirect jump prediction—with (a) perfect prediction, (b) intermediate level of prediction, and (c) no indirect jump prediction.

Window size—for some of the processor models, different window sizes from an upper limit of 2048 instructions down to 4 instructions were used.

Cycle width, i.e. the maximum number of instructions which can be issued in one cycle—(a) 64, (b) 128, and (c) bounded only by window size. Note that, from a practical point of view, cycle widths of both 64 and 128 are on the high side.

Latencies of processor operations—five different latency models were used, specifying latencies (in number of clock cycles) for various processor operations.

Loop unrolling—was carried out in some of the programs.

Misprediction penalty—values of 0 to 10 clock cycles were used.

Conclusions Reached With 18 programs and more than 350 processor configurations, it should come as no surprise to the student that Wall's research generated copious results. These results are presented systematically in the full report, which is available on the web. For our purposes, we summarize below the main conclusions of the report.

For the overall degree of parallelism found, the report says:

Using nontrivial but currently known techniques, we consistently got parallelism between 4 and 10 for most of the programs in our test suite. Vectorizable or nearly vectorizable programs went much higher.

Branch prediction and speculative execution is identified as the major contributor in the exploitation of instruction level parallelism:

Speculative execution driven by good branch prediction is critical to the exploitation of more than modest amounts of instruction-level parallelism. If we start with the Perfect model and remove branch prediction, the median parallelism plummets from 30.6 to 2.2 ...

Perfect model in the above excerpt refers to a processor which performs perfect branch prediction, jump prediction, register renaming, and alias analysis. The student will appreciate readily that this is an ideal which is impossible to achieve in practice.

Overall, Wall's study reports good results for the degree of parallelism; but the report also goes on to say that the results are based on '*rather optimistic assumptions*'. In the actual study, this meant: (i) as many copies of functional units as needed, (ii) a perfect memory system with no cache misses, (iii) no penalty for missed predictions, and (iv) no speed penalty of the overhead for aggressive pursuit of instruction level parallelism.

Clearly no real hardware processor can satisfy such ideal assumptions. After listing some more factors which lead to optimistic results, the report concludes:

Any one of these considerations could reduce the expected payoff of an instruction-parallel machine by a third; together they could eliminate it completely.

The broad conclusion of Wall's research study therefore certainly seems to support the proverb quoted at the start of this section.

In Chapter 13, we shall review recent advances in technology which have had a major impact on processor design, and we shall also look at some specific commercial products introduced in recent years. We shall see that the basic techniques and trade-offs discussed in this chapter are reflected, in one form or another, in the processors and systems-on-a-chip introduced in recent years.

Table 12.1 *Programs included in Wall's study*

Name of program	*Function performed*	*No. of instructions executed (millions)*
sed	Stream editor	1.46
egrep	File search	13.72
yacc	Compiler-compiler	30.29
metronome	Timing verifier	71.27
Grr	PCB router	144.44
Eco	Recursive tree comparison	27.39
gcc1	First pass GNU C compiler	22.75
Espresso	Boolean function minimizer	134.43
Li	Lisp interpreter	263.74
Fpppp	Quantum chemistry benchmark	244.27
Doduc	Hydrocode simulation	284.42
Tomcatv	Vectorized mesh generation	301.62
hydro2d	Astrophysical simulation	8.23
Compress	Lempel-Ziv file compression	88.27
Ora	Ray tracing	212.12
swm256	Shallow water simulation	301.40
alvinn	Neural network training	388.97
mdljsp2	Molecular dynamics model	393.07

12.12 THREAD LEVEL PARALLELISM

We have already seen that dependences amongst machine instructions limit the amount of instruction level parallelism which is available to be exploited within the processor. The dependences may be true data dependences (RAW), control dependences introduced by conditional branch instructions, or resource dependences[13].

One way to reduce the burden of dependences is to combine—with hardware support within the processor—instructions from multiple independent threads of execution. Such hardware support for multi-threading would provide the processor with a pool of instructions, in various stages of execution, which have a relatively smaller number of dependences amongst them, since the threads are independent of one another.

Let us consider once again the processor with instruction pipeline of depth eight, and with targeted superscalar performance of four instructions completed in every clock cycle (see Section 12.11). Now suppose that these instructions come from four independent threads of execution. Then, on average, the number of instructions in the processor at any one time from one thread would be $4 \times 8/4 = 8$.

With the threads being independent of one another, there is a smaller total number of data dependences amongst the instructions in the processor. Further, with control dependences also being separated into four threads, less aggressive branch prediction is needed.

Another major benefit of such hardware-supported multi-threading is that pipeline stalls are very effectively utilized. If one thread runs into a pipeline stall—for access to main memory, say—then another thread makes use of the corresponding processor clock cycles, which would otherwise be wasted. Thus hardware support for multi-threading becomes an important latency hiding technique.

To provide support for multi-threading, the processor must be designed to switch between threads—either on the occurrence of a pipeline stall, or in a round robin manner. As in the case of the operating system switching between running processes, in this case the hardware context of a thread within the processor must be preserved.

But in this case what exactly is the meaning of the *context of a thread*?

Basically, thread context includes the full set of registers (programmable registers and those used in register renaming), PC, stack pointer, relevant memory map information, protection bits, interrupt control bits, etc. For N-way multi-threading support, the processor must store at one time the thread contexts of N executing threads. When the processor switches, say, from thread A to thread B, control logic ensures that execution of subsequent instruction(s) occurs with reference to the context of thread B.

Note that thread contexts need not be saved and later restored. As long as the processor preserves within itself multiple thread contexts, all that is required is that the processor be able to switch between thread contexts from one clock cycle to the next.

As we saw in the previous section, there are limits on the amount of instruction level parallelism which can be extracted from a single stream of executing instructions—i.e. a single thread. But, with steady advances in VLSI technology, the aggregate amount of functionality that can be built into a single chip has been growing steadily.

[13] As discussed above, we assume that WAR and WAW dependences can be handled using some form of register renaming.

Therefore hardware support for multi-threading—as well as the provision of multiple processor cores on a single chip—can both be seen as natural consequences of the steady advances in VLSI technology. Both these developments address the needs of important segments of modern computer applications and workloads.

Depending on the specific strategy adopted for switching between threads, hardware support for multi-threading may be classified as one of the following:

(i) *Coarse-grain multi-threading* refers to switching between threads only on the occurrence of a major pipeline stall—which may be caused by, say, access to main memory, with latencies of the order of a hundred processor clock cycles.

(ii) *Fine-grain multi-threading* refers to switching between threads on the occurrence of any pipeline stall, which may be caused by, say, L1 cache miss. But this term would also apply to designs in which processor clock cycles are regularly being shared amongst executing threads, even in the absence of a pipeline stall.

(iii) *Simultaneous multi-threading* refers to machine instructions from two (or more) threads being issued in parallel in each processor clock cycle. This would correspond to a multiple-issue processor where the multiple instructions issued in a clock cycle come from an equal number of independent execution threads.

With increasing power of VLSI technology, the development of multi-core *systems-on-a-chip* (SoCs) was also inevitable, since there are practical limits to the number of threads a single processor core can support. Each core on the Sun UltraSpare T2, for example, supports eight-way fine-grain multi-threading, and the chip has eight such cores. Multi-core chips promise higher net processing performance per watt of power consumption.

Systems-on-a-chip are examples of fascinating design trade-offs and the technical issues which have been discussed in this chapter. Of course, we have discussed here only the basic design issues and techniques. For any actual task of processor design, it is necessary to make many design choices and trade-offs, validate the design using simulations, and then finally complete the design in detail to the level of logic circuits.

Over the last couple of decades, enormous advances have taken place in various areas of computer technology; these advances have had a major impact on processor and system design. In the next chapter, we shall discuss in some detail these advances and their impact on processor and system design. We shall also study in brief several commercial products, as case studies in how actual processors and systems are designed.

Summary

Processor design—or the choice of a processor from amongst several alternatives—is the central element of computer system design. Since system design can only be carried out with specific target application loads in mind, it follows that processor design should also be tailored for target application loads. To satisfy the overall system performance criteria, various elements of the system must be balanced in terms of their performance—i.e. no element of the system should become a performance bottleneck.

One of the main processor design trade-offs faced in this context is this: Should the processor be designed to squeeze the maximum possible parallelism from a single thread, or should processor hardware support multiple independent threads, with less aggressive exploitation of instruction level parallelism within each thread? In this chapter, we studied the various standard techniques for exploiting instruction level parallelism, and also discussed some of the related design issues and trade-offs.

Dependences amongst instructions make up the main constraint in the exploitation of instruction level parallelism. Therefore, in its essence, the problem here can be defined as: executing a given sequence of machine instructions in the smallest possible number of processor clock cycles, while respecting the true dependences which exist amongst the instructions. To study possible solutions, we looked at two possible prototype processors: one provided with a reorder buffer, and the other with reservation stations associated with its various functional units.

In theory, compiler-detected instruction level parallelism should simplify greatly the issues to be addressed by processor hardware. This is because, in theory, the compiler would do the difficult work of dependence analysis and instruction scheduling. Processor hardware would then be 'dumb and fast'—it would simply execute at a high speed the machine instructions which specify parallel operations. However, many types of runtime events—such as interrupts and cache misses—cannot be predicted at compile time. Processor hardware must therefore provide for dynamic scheduling to exploit instruction level parallelism, limiting the value of what the compiler alone can achieve.

Operand forwarding is a hardware technique to transfer a required operand to multiple destinations in parallel, in one clock cycle, over the common data bus—thus avoiding sequential transfers over multiple clock cycles. To achieve this, it is necessary that processor hardware should dynamically detect and exploit such potential parallelism in data transfers. The benefits lie in reduced wait time in functional units, and better utilization of the common data bus, which is an important hardware resource.

A reorder buffer is a simple mechanism to commit instructions in program order, even if their corresponding operations complete out of order within the processor. Within the reorder buffer, instructions are queued in program order with four typical fields for each instruction—instruction identifier, value computed, program-specified destination of the value computed, and a flag indicating whether the instruction has completed. This simple technique ensures that program state and processor state are correctly preserved, but does not resolve WAR and WAW dependences within the instructions.

Register renaming is a clever technique to resolve WAR and WAW dependences within the instruction stream. This is done by re-mapping source and target programmable registers of executing machine instructions to a larger set of program-invisible registers; thereby the second instruction of a WAR or WAW dependence does not write to the same register which is used by the first instruction. The renaming is done dynamically, without any performance penalty in clock cycles.

Tomasulo's algorithm was developed originally for the IBM 369/91 processor, which was designed for intensive scientific and engineering applications. Operand forwarding is achieved using source tags, which are also sent on the common data bus along with the operand value. Use of reservation stations with functional units provides an effective register renaming mechanism which resolves WAR and WAW dependences.

About 15% to 20% of instructions in a typical machine language program are branch and jump instructions. Therefore for any pipelined processor—but especially for a superscalar processor—branch

prediction and speculative execution are critical to achieving targeted performance. A simple two-bit counter for every branch instruction can serve as a basis for branch prediction; using a combination of local and global branch prediction, more elaborate schemes can be devised.

Limitations in exploiting greater degree of instruction level parallelism arise from the increased overhead in the required control logic. The limitations may apply to achievable clock rates, power consumption, or the actual processor utilization achieved while running application programs. Thread-level parallelism allows processor resources to be shared amongst multiple independent threads executing at one time. For the target application, processor designers must choose the right combination of instruction level parallelism, thread-level parallelism, and multiple processor cores on a chip.

Exercises

Problem 12.1 Define in brief the meaning of *computer architecture*; within the scope of that meaning, explain in brief the role of processor design.

Problem 12.2

(a) When can we say that a computer system is *balanced* with respect to its performance?

(b) In a particular computer system, the designers suspect that the *read/write* bandwidth of the main memory has become the performance bottleneck. Describe in brief the type of test program you would need to run on the system, and the type of measurements you would need to make, to verify whether main memory bandwidth is indeed the performance bottleneck. You may make additional assumptions about the system if you can justify the assumptions.

Problem 12.3 Recall that, in the example system shown in Fig. 12.1, the bandwidth of the shared processor-memory bus is a performance bottleneck. Assume now that this bandwidth is increased by a factor of six. Discuss in brief the likely effect of this increase on system performance. After this change is made, is there a likelihood that some other subsystem becomes the performance bottleneck?

Problem 12.4 Explain in brief some of the basic design issues and trade-offs faced in processor design, and the role of VLSI technology selected for building the processor.

Problem 12.5 Explain in brief the significance of (i) *processor state*, (ii) *program state*, and (iii) *committing* an executed instruction.

Problem 12.6

(a) Explain in brief, with one example each, the various types of dependences which must be considered in the process of exploiting instruction level parallelism.

(b) Define in brief the problem of exploiting instruction level parallelism in a single sequence of executing instructions.

Problem 12.7 With static instruction scheduling by the compiler, the processor designer does not need to provide for dynamic scheduling in hardware. Is this statement true or false? Justify your answer in brief.

Problem 12.8 Describe in brief the structure of the reorder buffer, and the functions which it can and cannot perform in the process of exploiting instruction level parallelism.

Note for Exercises 9 to 15

The following three sequences of machine instructions are to be used for Exercises 9 to 15. Note that instructions other than LOAD and STORE have three operands each; from left to right they are, respectively, source 1, source 2 and destination. '#' sign indicates an immediate operand.

Assume that (a) one instruction is issued per clock cycle, (b) no resource constraints limit instruction level parallelism, (c) floating point operations take two clock cycles each to execute, and (d) *load/store* memory operations take one clock cycle each when there is L1 cache hit.

```
Sequence 1:   1     LOAD     mem-a, R1
              2     LOAD     mem-b, R2
              3     LOAD     mem-c, R3
              4     FADD     R2, R1, R1
              5     FSUB     R3, R1, R1
              6     STORE    mem-a, R1

Sequence 2:   1     LOAD     mem-a, R1
              2     FADD     R2, R1, R1
              3     STORE    mem-a, R1
              4     FADD     #1, R3, R2
              5     STORE    mem-d, R2
              6     LOAD     mem-e, R2

Sequence 3:   1     LOAD     mem-a, R1
              2     FADD     R1, R2, R2
              3     STORE    mem-b, R2
              4     FADD     #1, R3, R2
              5     STORE    mem-d, R2
              6     LOAD     mem-e, R2
```

Problem 12.9 Draw dependence graphs of the above sequences of machine instructions, marking on them the type of data dependences, with the respective registers involved.

Problem 12.10 Assume that the processor has no provision for register renaming and operand forwarding, and that all memory references are satisfied from L1 cache. Determine the number of clock cycles it takes to execute the above sequences of instructions, counting from the last clock cycle of instruction 1.

Problem 12.11 Now assume that register renaming is implemented to resolve WAR and WAW dependences. Determine the number of clock cycles it takes to execute the above sequences of instructions, counting from the last clock cycle of instruction 1.

Problem 12.12 Comment on the scope for operand forwarding within the sequences of instructions. Assume that the *load/store* unit can also take part in operand forwarding.

Problem 12.13 Assume that, in addition to register renaming, operand forwarding is also implemented as discussed in Exercise 12. Determine the number of clock cycles it takes to execute the above sequences of instructions, counting from the last clock cycle of instruction 1.

Problem 12.14 Consider your answers to Exercises 10, 11 and 13 above. Explain in brief how these answers would be affected if an L1 cache miss occurs in instruction 1, which takes five clock cycles to satisfy from L2 cache.

Problem 12.15 With reference to Exercise 13, describe in brief how Tomasulo's algorithm would implement register renaming and operand forwarding.

Problem 12.16 Explain in brief the meaning of *alias analysis* as applied to runtime memory addresses.

Problem 12.17 A particular processor makes use of a 2-bit predictor for each branch. Based on a program execution trace, the actual branch behavior at a particular conditional branch instruction is found to be as follows:

$$\underbrace{T\ T\ \ldots\ T}_{k \text{ times}}\ N\ \underbrace{T\ T\ \ldots\ T}_{k \text{ times}}\ N\ \ldots$$

Here T stands for branch taken, and N stands for branch not taken. In other words, the actual branch behavior forms a repeating sequence, such that the branch is taken k times (T), then not taken once (N).

With the 2-bit branch predictor, find fraction of correct branch predictions made if $k = 1, k = 2, k = 5$ and $k = 50$

Problem 12.18 Discuss in brief the difference between *local* and *global* branch prediction strategies, and how a two-bit selector may be used per branch to select between the two.

Problem 12.19

(a) Wall's study on instruction level parallelism is based on *oracle-driven trace-based simulation.* Explain in brief what is meant by this type of simulation.

(b) Wall's study of instruction level parallelism makes certain 'optimistic' assumptions about processor hardware. What are these assumptions? Against each of these assumptions, list the corresponding 'realistic' assumption which we should make, keeping in view the characteristics of real processors.

Problem 12.20 Discuss in brief the basic trade-off in processor design between exploiting instruction level parallelism in a single executing thread, and providing hardware support for multiple threads.

Problem 12.21 Describe in brief what is mean by the *context of a thread,* and what are the typical operations involved in switching between threads.

Problem 12.22 Describe in brief the different strategies which can be considered for switching between threads in a processor which provides hardware support for multi-threading.