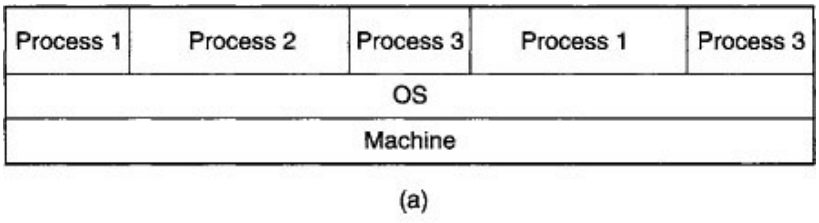
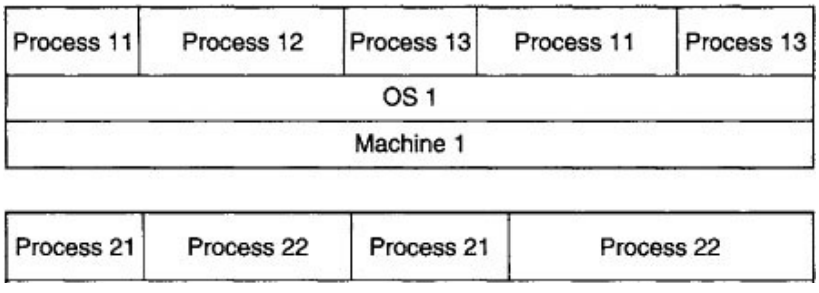
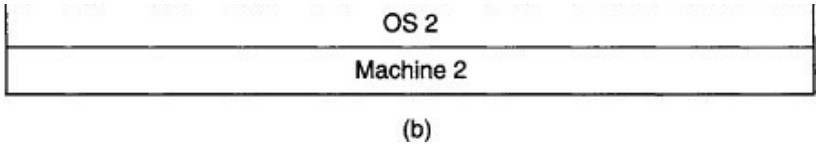
Thus there is a performance degradation of a process due to this sharing of resources with other processes, though this degradation can be reduced by scheduling the context switch at points in the computation where the process would have to wait for some external event such as I/O. Figure A.15a shows the multiplexing of processes in a conventional uniprocessor multiprogrammed system with a single hardware processor and operating system. The multiple user processes run on the same OS, which time-multiplexes the processor and other hardware resources among these processes.

Multiprocessors, on the other hand, employ multiple processors to achieve parallel execution of processes. The multiplicity of program counters, hardware registers, execution units, caches, and sometimes even memory are used to handle multiple processes. If each process is designed to run on a single processor, there is no degradation of performance of the process as a result of executing other processes on the other processors. However, there are programs that are designed in such a way that the multiple processes running on the different processors need to communicate with each other, for example, to share data values or to signal events. The performance of the program is often determined by the efficiency of implementation of this communication.



(a)





(b)

Figure A.15 Comparison of Activity over Time in a Multiprogrammed System and a Clustered Multiprocessing System. *(a) In multiprogramming, several tasks or processes running under a single operating system are multiplexed in time. (b) In multiprocessing, the processes are distributed among multiple processors; however, the number of operating systems cannot exceed the number of processors.*

Further parallelism in execution can be obtained by implementing multiprogramming on each of the processors of a multiprocessor system, as shown in Figure A.15b. The figure illustrates the case where an operating system and its application processes are dedicated to a specific processor in the multiprocessing system. It is also possible to design the multiprocessor with just one operating system that controls the allocation of processes to processors. In this case, a process may run on one processor at one time and on another at another time.

Historically, many systems have been designed around a single processor, and such uniprocessor systems remain an important system class. Today, however, multiprocessors are in widespread use as servers and are well on their way to becoming commonplace for high-end desktop computing. For server implementations, multiprocessor systems with large main memories, disk systems, and high-bandwidth network facilities provide important economies of scale as well as improved opportunities for balancing resources among a large number of simultaneously active application programs. System software employs systemwide data structures for managing processor, memory, and communication resources. When an application thread is initiated, system software assigns it resources from the pool of available processors, and, as it runs, system software provides it with memory, disk, and communication resources.

A.7.1 Types of multiprocessing

There are two main paradigms for supporting multiprocessing: *clus-*

tered computing and *shared-memory multiprocessing.* In many server environments, e.g., Web or database servers, the individual application processes are often independent of each other. This type of workload works well on clustered systems, where the hardware associated with each processor, including the memory and sometimes the disks, is separated from the hardware of other processors on the system. System software in this case typically consists of different copies of an operating system running on each processor, as shown in Figure A.15b.

There are other multiprocessor applications where a single program actually consists of a number of closely cooperating parallel processes. Many scientific and engineering applications are multithreaded in this way, as are compute-intensive commercial applications such as data mining. Many other commercial database applications also are structured with multiple processes sharing data, and they are best implemented on a multiprocessor system with a single operating system. The operating system manages the memory in the system as one large structure shared symmetrically among all the processors and hence the name *shared-memory processing,* or SMP. The shared memory acts like a bulletin board and may be used either to pass data values from one process in the system to another or to signal the occurrence of an event to other processors that may be waiting for that event to occur. Reliable management of this communication of data and events needs special synchronization primitives to be supported by the ISA. The two types of multiprocessor systems are illustrated in Figure A.16.

There also are systems that are hybrids of the clustered and shared-memory forms of multiprocessing. For example, *distributed shared memory* (DSM) systems are implemented like clustered systems but are capable of supporting a single operating system image across the multiple processors. It is hard to achieve as low a latency of communication on such a system as on a shared-memory system. A lot of research has been conducted in this area, and several differ-

lot of research has been conducted in this area, and several different hardware and software techniques have been proposed to improve or hide the communication latency. But this form of computing did not gain popularity, possibly because of the high sensitivity of the performance of applications to the way the algorithms are written and data structures are laid out in memory.

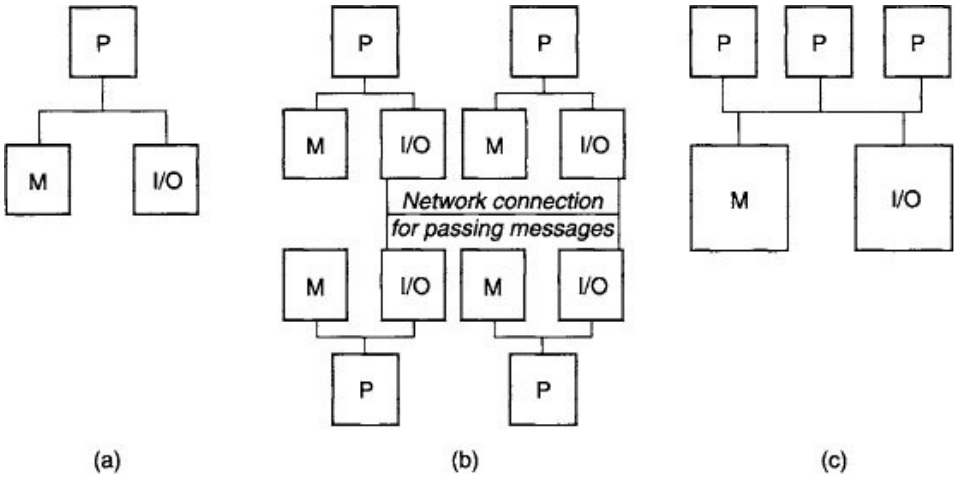


Figure A.16 Comparison of Uniprocessor and Two Common Multiprocessor Systems. *(a) Uniprocessor; (b) message-passing multiprocessor system; (c) shared-memory multiprocessor system (P = processor, M = memory).*

On the other hand, a different type of hybrid appears to be emerging as an important player—*SMP clusters*. Many server systems today are clusters of nodes, where each node itself is a small shared-memory multiprocessor. Communication between processors in a node is through shared memory, while communication between nodes is through messages. Unlike the distributed shared-memory paradigm, there is no effort made in hardware to improve the communication latency between nodes; consequently, operating systems are not expected to span multiple nodes in an SMP cluster.

A.7.2 Clustered Systems

The most common form of cluster is simply a network of computers. Users access the cluster through *terminals*. Terminals may be simply devices that allow the user to type in requests and receive information to be displayed on a screen. More commonly, terminals are

complete workstations or PCs comprising processors, memory, disks, and peripherals. When connected to a network, the terminal can access data located at other nodes in the network. The network itself may be spread over a large area or may be quite compact, with high-speed interconnections. A degree of isolation is achieved on a cluster by the fact that applications generally run on the processor at a local node and cannot easily be affected by applications running on other nodes in the network.

Besides the terminal nodes just described, the network in a cluster typically also has various types of servers attached to it. We have already seen the need for file servers that are repositories of data for users of a cluster. There can also be Web servers, which are nodes that serve as access points to the World Wide Web, mail servers, which are nodes that provide mail services, acting as a repository for new and old mail, and print servers, which service printing jobs from other nodes.

Clustering is also common in very large server systems. Here each node in a cluster may be a large shared-memory multiprocessing system, for example, a 32-way (processor) or 64-way system. Multiple nodes of this type are interconnected by a fast, high-bandwidth, optical network, as shown in Figure A.17. In the largest of such systems, the high-bandwidth interconnection may be a switch rather than a bus. Examples of such clusters are the IBM Parallel Sysplex system for the mainframes (**Nick et al. 1997**) or the HP Superdome Hyperplex system (**Charlu 1999**). Originally these systems provided a solution to expand beyond the capabilities of a single box. Their attractive features include a suite of software system management tools that allow workload balancing and graceful degradation from failures. Additionally, communication routines are provided to enable an application to use resources associated with multiple nodes in a cluster.

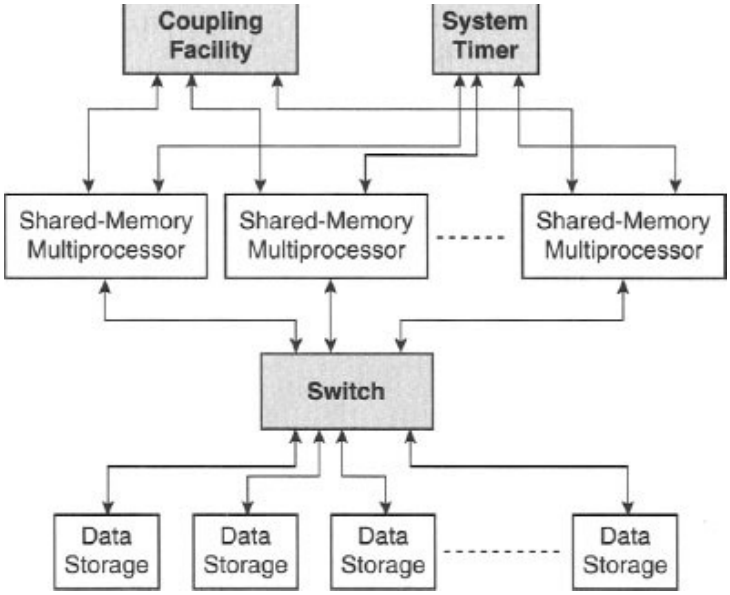


Figure A.17 Typical Large Database Cluster. *The coupling facility provides hardware support for synchronization and coherence between nodes of the cluster. The system timer ensures that the clocks on all nodes are synchronized. The data storage, typically on large disks, is shared between nodes and accessed through a high-bandwidth fiber-optic switch.*

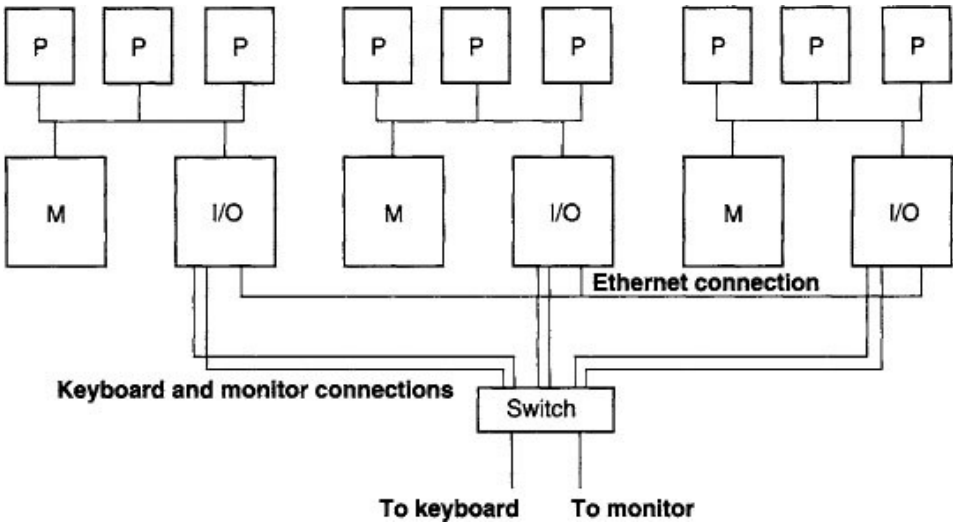


Figure A.18 A Cluster of SMP Nodes. *Beowulf clusters are typically built using inexpensive off-the-shelf tower-type desktop computers connected together through Ethernet and sharing a single monitor and keyboard for the use of the system administrator.*

The availability of inexpensive commodity uniprocessor systems has led to various architectures that combine commodity components to build inexpensive multiprocessor systems. The nodes in some cluster systems, such as the Beowulf cluster systems (**Ridge et al. 1997**), are off-the-shelf uniprocessor or small-scale SMP system boxes with off-the-shelf disks, connected together through commod-

ity networks such as 100Mb/sec Ethernet and running an inexpensive (or free) operating system such as Linux (see Figure A.18). A denser form of the Beowulf cluster, called the *blade server*, is rapidly gaining popularity. Blade servers essentially sacrifice I/O expandability, eliminate the monitor, keyboard, case, and power supply of desktop boxes, and put the uniprocessor or SMP subsystem in a thin package called a *blade*. These blades are then slid into standard racks, where a backplane provides the Ethernet connection and other wiring as well as power.

From a programming point of view, the distinction between shared-memory and clustered types of multiprocessor systems arises from the way processes communicate with each other. In shared-memory systems, two processes communicate by writing to or reading from memory locations accessible to both. In clustered systems, processes communicate by sending messages, and hence they are also referred to as *message-passing* multiprocessor systems. Unlike a shared-memory system, a clustered system presents more of a challenge to an application programmer. In both cases, the programmer is required to organize and partition a program into threads of computation, each to be executed on a different processor. In the cluster case, the programmer is also responsible for the task of synchronizing the computation using messages passed between the threads. Clustered systems would not have gained as much popularity as they have were it not for the software tools and libraries that have been introduced to help develop applications for such systems. Message passing has been recognized as a technique applicable to both closely coupled clusters and more loosely coupled distributed multiprocessor systems. This has led to the development of standard interfaces that allow a programmer to abstract the communication between processors in both types of environments. One of the more widely used interfaces of this type is the message-passing interface (MPI) (**Pacheco 1996**), which specifies a set of library routines that enable processes to communicate with each other and to exchange data. Portability of applications between systems that implement

MPI is one of the biggest strengths of the interface.

A.7.3 Evolution of Shared-Memory Systems

Many commercial applications benefit from having a shared-memory paradigm rather than a clustered design. The shared-memory paradigm is more convenient to reason with and to write programs for, but the hardware needed to support these systems is more complex and specialized as compared to those for message-passing systems. The complexity in hardware arises from the need to maintain coherence in memory locations, especially in modern memory hierarchies, where memory elements can reside at various levels of cache. Message-passing systems avoid this complexity by shifting this responsibility to the application programmer or to library routines.

The ease of programming shared-memory systems, along with innovative hardware techniques, has led to the development of larger and larger SMP systems over the last decade. Today, we find large shared-memory multiprocessor systems having 128 or more processors.

The ISA of most processors today includes special conventions for the access of shared-memory locations. As with all aspects of an ISA, these conventions represent a contract between the programmer and an implementation of the ISA. A programmer writes a program assuming that accesses to variables shared among processors obey certain rules, while the hardware designer (the processor designer or the system designer) ensures that the rules used by the programmer are indeed guaranteed by the implementation.

Memory Coherence Models

Memory coherence refers to the visibility of a write to a given memory location by all other processors in the system. Ideally, one would

like the value written to a memory location to be instantaneously visible to all other processors, meaning that a subsequent read to that location by any other processor should see the new value. This "instantaneous" solution is impractical, however, because of the several layers in the memory hierarchy of most modern systems. So a more practical definition of coherence is adopted—*memory coherence* is said to be implemented on a multiprocessor system if the order of writes to a given location by one processor is maintained when observed by any other processor in the system. Notice that this definition gives flexibility to a hardware designer to delay the observation of changes made by one processor on the other processors. Over the years, there have been several implementation protocols developed to facilitate coherence. These protocols are largely implementation alternatives and are typically not part of an ISA specification.

Consider the example of Figure A.19. Assume that processors 1 and 2 both have write-through caches. When each processor writes to location 50, it does so to a cached version of location 50. As a result, a read following a write will reflect the value in the cached version of location 50. Because the caches are write-through, the value gets written out to memory, though it takes longer for processor 2's write request to reach the memory. In the meantime, the cached value in processor 2 is evicted. When the value of location 50 is requested the next time, both processors get value 1. After a sufficiently long time, when processor 2's write reaches memory and neither processor has a cached copy of location 50, both processors read value 2 from memory. Thus, assuming a starting value of 0, processor 1 sees a transition from 0 to 1 to 2 in location 50, while processor 2 sees the transition from 0 to 2 to 1 to 2 for the same location. This is a violation of the memory coherence rule because it would never be possible for processor 1 to see the 0–2–1–2 order of values observed for processor 2. We note here that such a situation would not have occurred either on a multiprogrammed system or on a multiprocessor

system having instantaneous access to memory—models that most programmers are comfortable working with

There are several ways to avoid such memory coherence violations. One way is to ensure that when a value is written to the cache of a processor, copies of that location cached in all other processors are invalidated and, additionally, to ensure that any read request by a processor is satisfied from a cache that has the most recent valid copy, if the memory itself has not yet received the valid copy. The reader is referred to **Culler and Singh (1999)** for a study of various cache coherence protocols that have been designed to ensure memory coherence.

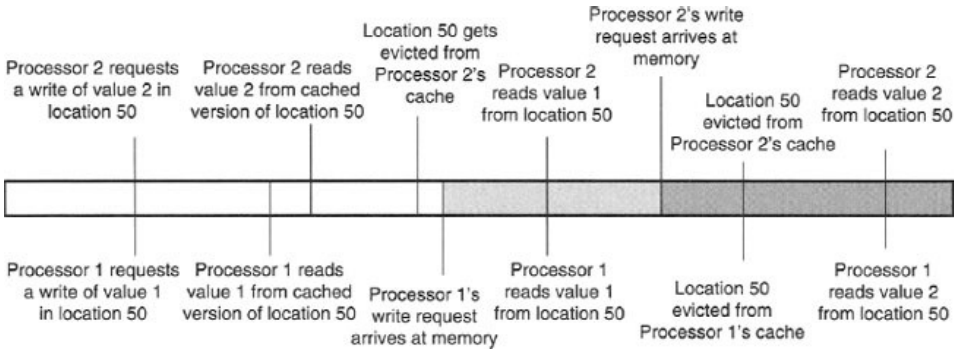


Figure A.19 Violation of Memory Coherence. *The bottom part of the figure shows a write followed by three reads of location 50 by processor 1, while the top part shows the same sequence for processor 2. The shaded region indicates the time when the memory contains value 1, while the hatched region indicates the time when it contains value 2. Note that the values read from the caches are different from the values in memory. The sequence of values returned for processor 1 due to reads is 0–1–1–2, while that returned for processor 2 is 0–2–1–2.*

Memory Consistency Models

Memory consistency characterizes the order in which accesses by one processor to different locations in memory are observed by another processor. This is in contrast to memory coherence, which deals with the order of writes to a single location by a processor. Memory consistency concerns itself not only with accesses to different locations but with both types of accesses, reads, and writes. In general, the more constraining the consistency rules are, the easier it is for a programmer to develop and debug a program but the

harder it is for a hardware designer to produce a high-performance implementation of the processor. An implementation that forces all processors to wait until a written value is visible to all processors in the system provides a maximum level of consistency but is uninteresting because of its high overhead. The frequent, and long, waits would negate a lot of the parallelism benefits expected from executing the application on such a multiprocessor system.

The *sequential consistency* model, described first by Leslie **Lamport (1979)**, is an elegant and natural model. A multiprocessor system maintains sequential consistency if the set of observable memory access orderings made by a multithreaded program is a subset of the observable orderings when the program is run on a multiprogrammed uniprocessor system. It is a natural definition, and it is restrictive but still flexible enough to allow the exploitation of most of the benefits from multiprocessing. Even in a multiprogrammed execution of a multithreaded program, the set of observable access orderings is large, with the variation in ordering arising from the variation of points in time that context switches occur and the variation in allocation of waiting threads to the processor. This is illustrated in Figure A.20.

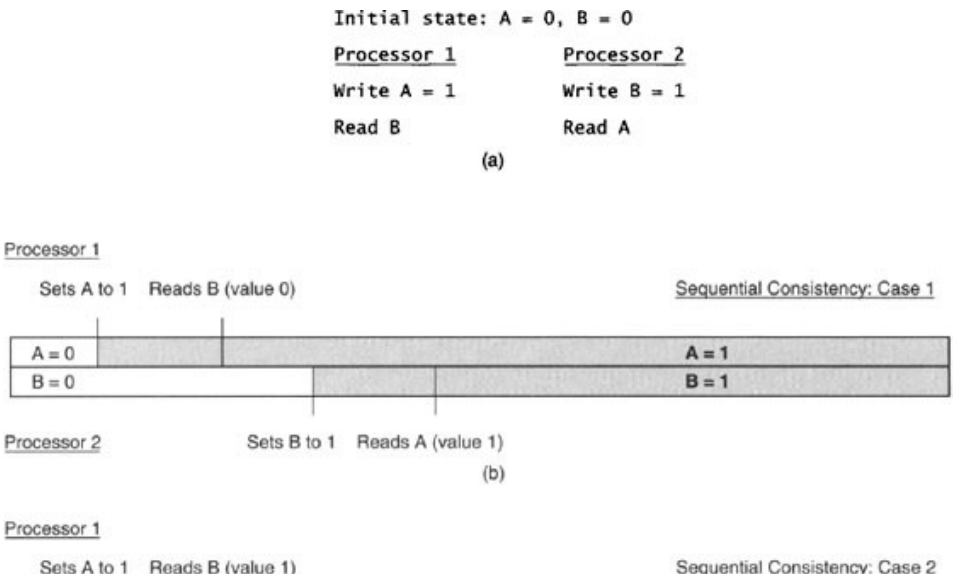
We will simply state here, without proof, that a sufficient condition for sequential consistency in a multiprocessor system is the following: For every pair of accesses to memory by any one processor, the first access in program order is observed by all other processors before the second access is observed. Any implementation that satisfies this condition guarantees sequential consistency in the system.

Processor designers are always looking for ways to improve the single-thread performance in their implementations. Modern superscalar implementations attempt to increase throughput by performing independent operations in the instruction stream in an order that may be different from the original program order. Sequential consistency can get in the way of designing fast uniprocessor imple-

mentations. The general philosophy in these implementations is that programmers generally know the exact points in their programs where they would like ordering preserved, and hence it is not necessary to force an implementation to maintain sequential consistency always. Let us now examine some possible relaxations.

When a pair of memory accesses is made by a processor, various conditions in the hardware, specifically the latency of completing these accesses, may cause the second operation to appear to occur before the first. This potential violation of pair order is referred to as a *hazard.* There are four kinds of possible hazards.

- **A read—read (RR) hazard:** This means that if there are two reads that occur in program order, the second read is seen to occur after the first read. One would assume that when there are no intervening writes, such a reordering of reads should not pose any problem. This is indeed the case with single-threaded programs. However, in multithreaded programs on a multiprocessor system, there is the possibility that a processor accessing one of these locations for writing causes the values read to be different from the values read in an alternative implementation that forces a read-to-read ordering.
- **A write—read (WR) hazard:** This refers to a reordering of memory accesses such that a read occurs before a write, even though program order requires the read to be after the write. A processor is always designed to avoid such a hazard when the locations addressed by the write and read operations are identical.



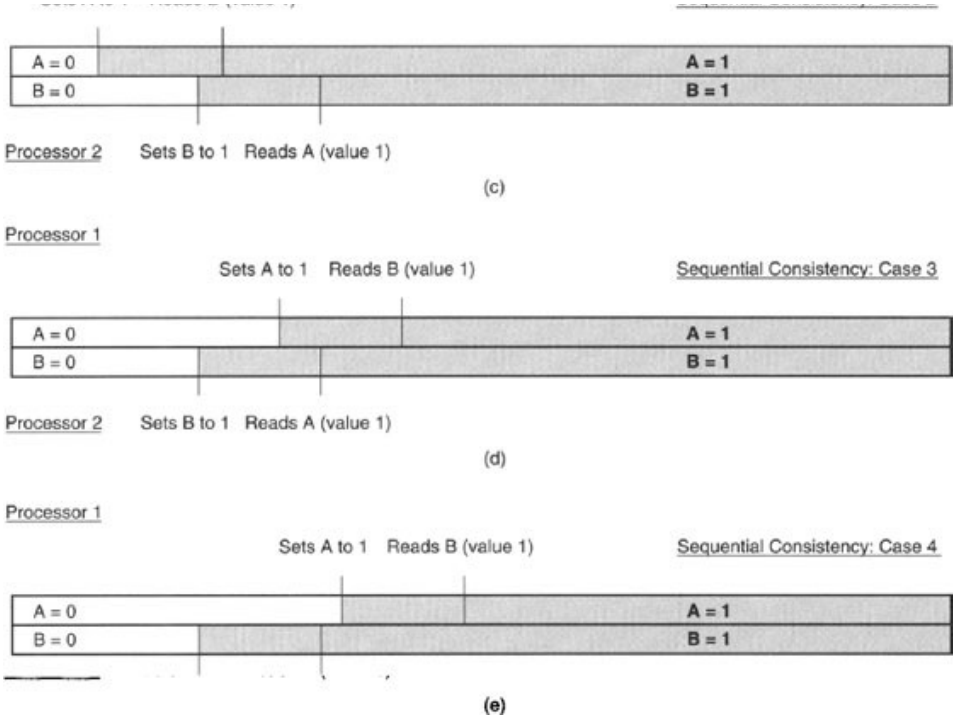


Figure A.20 Sequential Consistency. *If the program shown in (a) were executed in a multiprogrammed environment, the only possible results are those shown in (b) through (e). Hence any sequentially consistent system must produce one of these results. In particular, it is impossible for the values read to both be 0 at the end of these sequences.*

- **A read—write (RW) hazard:** This occurs when a write occurs before a read, even though program order requires the write to be after the read.
- **A write—write (WW) hazard:** Such a hazard occurs when two writes are performed in an order opposite to what was required in the program. Once again, correct execution on a single thread ensures that such a hazard cannot take place when the two referenced locations are identical.

An ISA that disallows all four types of hazards in any of its implementations is said to support a strong consistency model. Most processor ISAs specify consistency rules that are weaker than such a strong model. Both the IBM System/390 and the IA-32 processors originally specified one such relaxed ordering rule, referred to today as *processor consistency*. This rule allows the relaxation of the WR ordering; i.e., it permits operands of an instruction to be fetched even before the result of a previous instruction is written all the way to memory. In the absence of such a relaxation, pipelining of operations in a processor would be seriously hampered because instructions would not be allowed to start access of operands even when these operands are independent of the values written by im-

mediately preceding instructions.

Figure A.21 illustrates how a multiprocessor system implementing a processor consistency model can produce results that are not possible if the system followed the sequential consistency model. The program example is taken from Figure A.20. In the example, if both processors perform the reads before the writes, then it is possible for both processors to see old values simultaneously at the end of the program segment. On the other hand, as Figures A.20b through e illustrate, at least one of the processors must see a new value if the system implemented sequential consistency.

With the advent of superscalar processors that decouple memory access from instruction execution, memory models began to specify even weaker consistency rules. For example, *release consistency* (**Gharachorloo et al. 1990**), which is now supported by many modern architectures, relaxes all memory-ordering requirements except for explicit synchronization operations. The responsibility for correctness in the execution of a multithreaded program is thus shifted completely to the programmer. Special instructions, called *memory barrier instructions,* are provided to allow the programmer or the compiler to impose ordering relationships wherever desired. Examples of memory barrier instructions include the MEMBAR instruction in Sun's SPARC architecture and SYNC in the PowerPC architecture. Both of these ensure that the results of operations after the memory barrier are not seen before the results of operations that occur earlier than the memory barrier have been recorded. Thus, if a strict ordering is desired in the example in Figure A.21a, a memory barrier placed between the writes and the reads would prevent the load instructions from being completed before the results of the store instructions have been recorded.

Initial state: A = 0, B = 0

Processor 1	Processor 2
Write A = 1	Write B = 1

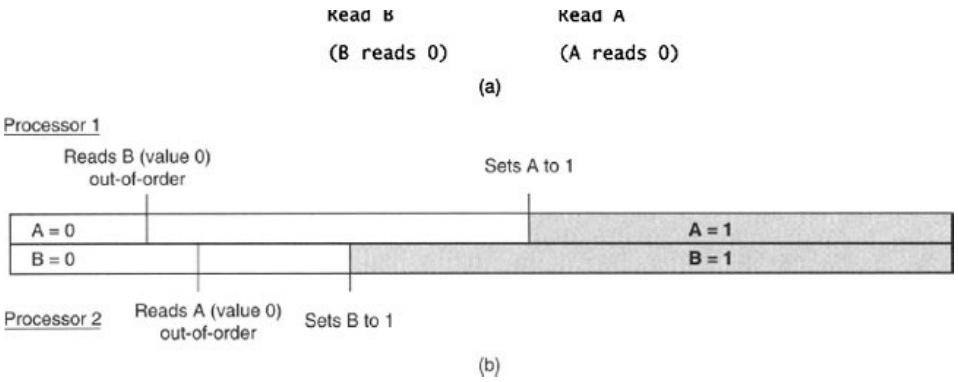


Figure A.21 Sample Program to Illustrate Processor Consistency. *In processor consistency, the reads are allowed to go ahead of preceding writes as long as they are not to the same location. If this occurs, the reads result in a value of 0 for both A and B. This combination is not one of the possibilities shown in Figures A.20b through e.*

A.8 Example Instruction Set Architectures

Most of the examples in this book involve the use of either the PowerPC ISA or the IA-32 ISA. Hence we will describe some of the salient characteristics of these two architectures in the rest of this appendix.

A.8.1 PowerPC ISA

The PowerPC architecture (**IBM 1994**) was defined as a joint effort by IBM, Apple, and Motorola in 1993. However, it was very closely derived from the earlier IBM POWER ISA. Currently PowerPC is used by IBM in its pSeries, eSeries, and iSeries lines of computers and by Apple in its Macintosh line. Versions of the PowerPC architecture are also used in embedded systems developed by IBM and Motorola, in the game machines sold by Nintendo, in the Sony-Toshiba-IBM (STI) Cell processor, and in the IBM BlueGene supercomputer.

The PowerPC ISA is an example of a RISC ISA. The original 32-bit ISA was extended to a 64-bit ISA because of the need to address larger memories and because of the performance advantages of operating

memories and because of the performance advantages of operating on larger units of data. We begin with a description of the state (registers and memory architecture) and then summarize the actual instruction set.

Registers

The PowerPC register set is illustrated in Figure A.22. There are 32 general-purpose registers and 32 floating-point registers. The floating-point registers are 64 bits wide, while the general-purpose registers are 32 bits or 64 bits, depending on the version of the architecture implemented. Both versions can be simultaneously supported in an implementation, in which case the physical registers are 64 bits wide, while only the lower 32 bits will be active when operating in the 32-bit mode.

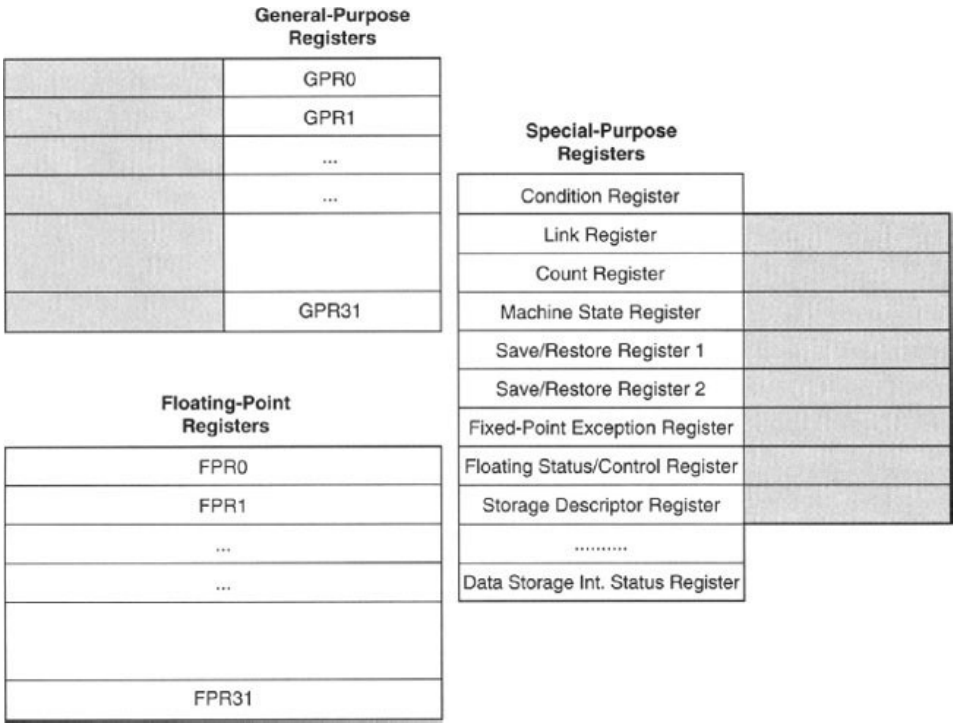


Figure A.22 PowerPC Register Set. *Shaded areas indicate registers that change size from 32 bits to 64 bits when going to 64-bit mode.*

The main special-purpose registers used by an application are the link register and the count register, each 32 or 64 bits wide, depending on the mode, and the 32-bit condition register, divided into eight

ing on the mode, and the 32-bit condition register, divided into eight independently addressable fields, one of which is set as a side effect of executing certain instructions. The fixed-point exception register contains fields set or used by fixed-point instructions, while the bits in floating-point status and control register allow the monitoring of certain conditions resulting from the execution of floating-point instructions. Both these registers contain "summary" bits that can be set implicitly by the execution of an instruction but must be reset explicitly. These bits provide summary information about whether a certain event occurred since they were last reset.

The remaining registers are generally not used during the compilation of a user program. These include a machine state register, MSR, which provides information about the current state of the machine, for example, whether it is in 32-bit mode or 64-bit mode, whether it is in privileged state, and whether address translation is on. There are two registers, SRR0 and SRR1, the machine state status and restore registers, respectively, that are used to save and restore information on interrupts. Other special-purpose registers include the data storage interrupt status register, DSISR, which provides information about the cause of storage and alignment interrupts, and a set of software-use registers SPGR0—3, specially designated for use by the operating system.

Memory Architecture

The address used by a PowerPC application is generally called the *effective address*. The term *virtual address* refers to a larger flat address space that includes the effective address spaces of all processes active on the system. Finally, the term *real address* is used in the conventional way to mean locations in real memory.

The PowerPC memory architecture implements segmentation, with segments that contain up to 2^{28} bytes (256MB). Pages are 2^{12} bytes (4KB) in size. The effective address space is 2^{32} bytes in 32-bit mode and 2^{64} bytes in 64-bit mode. The real address space is also limited

and 2^{64} bytes in 64-bit mode. The real address space is also limited to either 2^{32} bytes in 32-bit mode or 2^{64} bytes in 64-bit mode. The virtual address space is much larger, up to 2^{52} bytes in 32-bit mode or 2^{80} bytes in 64-bit mode.

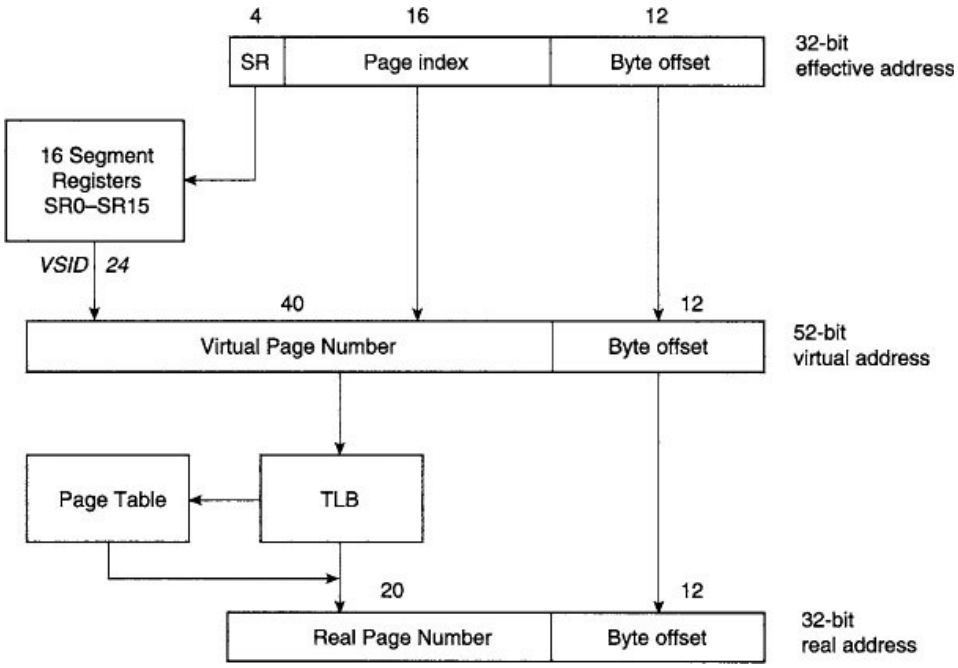


Figure A.23 Address Translation in the 32-bit PowerPC ISA.

In the 32-bit mode, there are 16 segment registers, SR0—15, that hold the locations in virtual memory of the 16 segments available in the effective address space. The top four bits of a 32-bit effective address point to one of the segment registers, which contains the top 24 bits of the segment address (refer to Figure A.23). This 24-bit virtual segment ID (VSID) along with the 16-bit page index from the effective address forms a 40-bit virtual page number (VPN). The VPN is then converted to a real page number (RPN) by referring to the page table whose location in real memory is determined using the storage descriptor register, SDR1.

The PowerPC has an architected page table; that is, the format of entries in the page table is defined in the ISA. In order to make references to the page table more efficient, the table is accessed associatively via hashing. The VPN is hashed in two different ways to arrive

at two pointers, each of which points to an entry group (PTEG) consisting of eight page table entries. A page fault occurs only if the required page cannot be found in either page table group. The probability of a page fault depends on various factors, including the hashing function used, the number of PTEGs implemented, and reference characteristics of the application.

The PowerPC architecture does not mandate the use of a translation lookaside buffer. However, it does provide instructions to maintain a TLB if one is implemented (as it always will be, practically speaking). For example, the **tlbie** instruction can be used to invalidate an entry in the page table when an entry is replaced in the hashed page table. All entries in the TLB can be invalidated (flushed) simultaneously using a **tlbia** instruction.

In 64-bit mode, the effective number of segments supported is 2^{36} instead of 16. The mapping of an effective segment ID to a virtual segment ID is done by means of a segment table pointed to by a special register called the *address space register* (ASR). Once again, in order to make accesses to this table more efficient, two hashed pointers are used to access two sets of segment table entry groups (STEGs), each set having eight segment table entries. A miss here causes a segment fault. A segment lookaside buffer (SLB) is used for avoiding a translation on each reference. Instructions (**slbie** and **slbia**) are provided to invalidate a single SLB entry or the entire SLB.

Instruction Set Summary

PowerPC uses a RISC instruction set that has fixed 32-bit-width instructions. All storage operations are conducted through load and store instructions. Thus there are no instructions that perform arithmetic or logical operations on the contents of memory. Any such operation requires that the operand be brought in from memory into one of the registers before being manipulated.

The PowerPC divides the user instruction set into three groups: the branch processor instructions, the fixed-point processor instructions, and the floating-point processor instructions.

The branch processor handles instructions that set or use the condition register, CR, or that change the flow of control modifying the program counter to deviate from the sequential execution of instructions. These latter instructions, the branch instructions, have one of the two following formats.

b	target	Branch to target address obtained by adding the address of the current instruction with the 24-bit offset specified in the instruction.
bc	cond, target	Branch conditionally to target obtained by adding the address of the current instruction with the 14-bit offset specified in the instruction. The condition for branching is determined by condition register bit BI and field BO in the instruction.

A useful variant of the branch instruction is the branch-and-link instruction. When the LK bit in the branch instruction is set, it indicates that a side effect of executing either the b or the bc instruction is to place the address of the immediately following instruction into LR, the link register. This is a useful way of saving the return address while branching to subroutines.

bl	target	Branch to target and save return address in link register.
bcl	target	Conditionally branch and link to target.

The branch processor includes instructions that allow various logical and transfer operations on individual bits or fields of the CR instruction. Another important instruction in the branch processor is the system-call instruction.

sc	Branch to a special location determined by the contents of the MSR, and save state information in SRR0 and SRR1. The processor enters system mode after executing this instruction.

The fixed-point processor executes instructions that involve use of the general-purpose registers, GPR0—31. Load and store instructions use the general-purpose registers to specify the address of the memory location involved and hence also involve the fixed-point processor. Here are examples of load and store instructions.

ld	rt, d(ra)	Load register rt with the contents of memory addressed by (ra) + d. The d field is treated as a signed quantity.
ldx	rt, ra, rb	Load register rt with the contents of memory addressed by (ra) + (rb).
lwz	rt, d(ra)	Load lower 32 bits of register rt from contents of memory addressed by (ra) + d; zero out the upper 32 bits of register rt.
lwzx	rt, ra, rb	Load lower 32 bits of register rt from contents of memory addressed by (ra) + (rb); zero out the upper 32 bits of register rt.
stw	rt, d(ra)	Store the 32 low-order bits of register rt in the memory location addressed by (ra) + d. The d field is treated as a signed quantity.
stwx	rt, ra, rb	Store the 32 low-order bits of register rt in the memory location addressed by (ra) + (rb).

The remaining fixed-point instructions are largely similar to those found in most RISC ISAs. The arithmetic and logical instructions operate on operands, both of which may be in general-purpose registers, or one of which may be a constant defined in the instruction. Instructions are nondestructive—the result register may be different from both operand registers. Examples follow.

add	rt, ra, rb	Add the contents of ra and rb and put the result in rt.
subf.	rt, ra, rb	Subtract the contents of ra from rb and put the result in rt. The dot in the mnemonic indicates that as a side effect of the instruction, bits in field 0 (first four bits) of the condition register must be set according to whether the result is negative, zero, or overflowed. This is called a *record-form instruction*, because it sets the record bit in the instruction.

Because branches are taken conditionally on the contents of the condition register, the compiler must either use a record form instruction or set any of the fields in the CR explicitly using a compare instruction. Here's an example.

cmpi	3, ra, rb	Compare the contents of registers ra and rb, treating them as unsigned integers, and set field 3 (bits 12–15) of the condition register according to whether ra is less than rb, rb is less than ra, or both are identical.

The PowerPC also has a rich set of rotate and shift instructions. These instructions optionally perform a mask operation during the shift process, which allows a wide variety of common operations in compilers and interpreters to be performed in a single instruction. An example is the following instruction.

rlwimi	ra, rs, sh, mb, me	Rotate the contents of rs left by the amount specified in sh, and merge the masked part into register ra. The part of ra that is changed is determined by a mask of 1's extending from the mb position to the me position.
rlwinm	ra, rs, sh, mb, me	Rotate the contents of rs left by the amount specified in sh, and use mask to extract a part of the rotated value to be transferred to ra.

A.8.2 The Intel IA-32 ISA

The IA-32 ISA (**Intel 1999**), informally known as the x86, descended from the Intel 8086 through the 286, 386, 486, and Pentium series. The 8086 started as a 16-bit microcontroller chip, primarily for use in embedded systems. An 8086 derivative, the 8088, was selected by IBM for the first PCs in 1981, and soon the x86 ISA evolved to a 32-bit general-purpose architecture. Recently, it has been further extended to 64 bits. Currently the IA-32 is probably the most widely used general-purpose architecture, with applications ranging from laptops to desktops to servers.

The IA-32 ISA is an example of a CISC ISA. The instruction set is more "cluttered" than "complex," there having been several instructions sets that are fundamentally more complex. The clutter comes from its evolutionary history and extensions, coupled with the need to maintain backward compatibility at each step. In this book we feature the 32-bit IA-32 architecture; the earlier 16-bit version of the ISA and the newer 64-bit version are not used in this book. Just as we did with the PowerPC description, we will begin with a descrip

we did with the PowerPC description, we will begin with a description of the state (registers and memory architecture) and then summarize the actual instruction set.

Registers

The IA-32 register set is illustrated in Figure A.24. There are eight general-purpose registers, six segment registers, and a floating-point stack with eight elements. The general-purpose registers are 32 bits wide, the segment registers are 16 bits wide, and the elements of the floating-point stack are 80 bits wide. For executing 16-bit instructions, the low-order 16 bits of the general-purpose registers are used. Some of the general-purpose registers have the special uses for certain instructions, some of which are listed below.

eax—Accumulator for operands and results

ebx—Pointer to data held in the DS segment

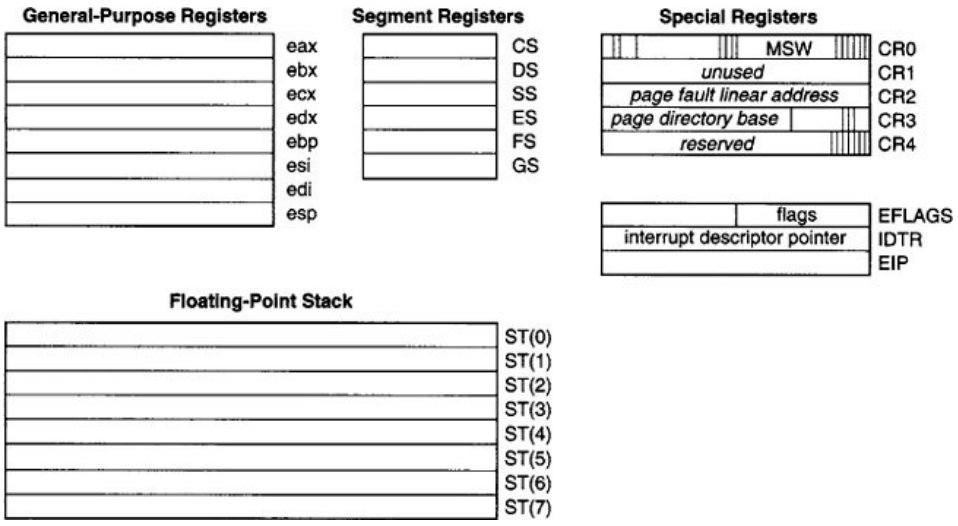


Figure A.24 IA-32 Register Set. *Registers shown are for 32-bit protected-mode operation.*

ecx—Counter for string operations and loops

edx—I/O pointer

esi—Pointer to data held in the DS segment; source pointer for string

esi—Pointer to data held in the DS segment; source pointer for string operations

edi—Pointer to data held in the ES segment; destination pointer for string operations

esp—Stack pointer in the SS segment

ebp—Pointer to data held in the stack

The special-purpose registers consist of five control registers, CR0—4, that contain a number of status and enable bits. For example, when a page fault occurs, CR2 is specified to hold the address causing the fault, while CR3 points to the page table. There are three other special-purpose registers, the EFLAGS, EIP, and IDTR registers. The EFLAGS register contains a number of status flags, including condition codes used by conditional branch instructions. The EIP is the program counter (referred to as the *instruction pointer* in Intel terminology). The IDTR, or the interrupt descriptor table register, points to a table in memory that holds interrupt vectors.

Memory Architecture

The IA-32 memory architecture (Figure A.25) is based on segments, somewhat similar to the PowerPC, but it is often used in a special configuration that presents a flat, linear model to software. The IA-32 ISA supports up to 64K segments, each of which can be up to 4GB. The segments map into a single, contiguous 32-bit linear address space (4GB), though at any given time only six of the segments can be addressed via six segment registers.

Like the PowerPC, a logical load or store address specifies one of the segment registers and an offset of up to 32 bits. Some instruction encodings can only specify one of the first four segments. The logical address points to a location in the linear 4GB address space. When the offset value is computed, typically by adding the contents of a

general-purpose register to a displacement value, the segment register is not changed. This is in contrast to PowerPC, where the segment pointer is in the upper bits of the effective address and can be affected by address arithmetic.

When used in its most general form, a segmented memory model can be constructed as shown in Figure A.25a. However, by setting all the segment registers to point to the base of the 32-bit linear address space, a simpler linear memory model can be constructed, as in Figure A.25b. It is this model that is used by most UNIX systems.

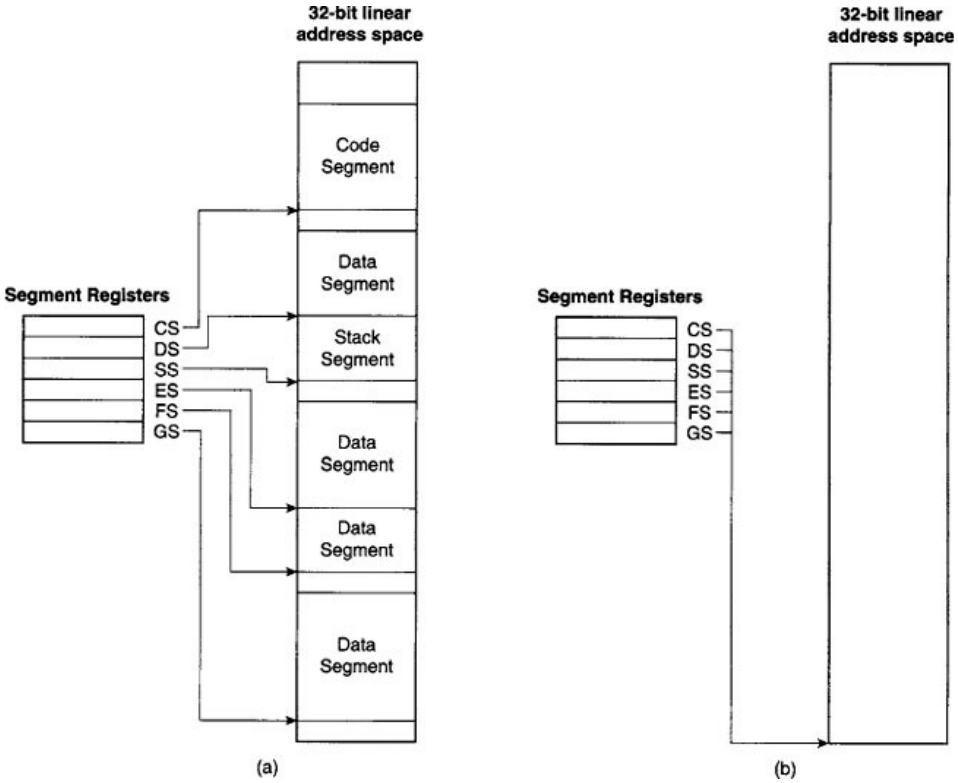


Figure A.25 Virtual memory Models Supported by IA-32. *(a) Segmented memory model; (b) flat memory model. The IA-32 also supports a real address model which is not shown.*

The linear address space is divided into 4KB pages, irrespective of the segments. The flat linear address space is then mapped onto a real memory that may have up to 4GB. As with the PowerPC, the Intel IA-32 has an architected page table, and hence the format of entries in the page table is defined in the ISA. The page table is similar to the one shown in Figure A.10, but it contains two levels rather

far to the one shown in Figure A.10, but it contains two levels rather than one. The 20-bit page number field of a virtual address is divided into an upper 10-bit directory pointer and a lower 10-bit page pointer. The directory pointer points to an entry in a 1K page directory. Each directory pointer points to a separate 1K entry page table, which is accessed using the lower 10-bit page pointer. The page directory is located at an address specified in CR3, which acts as the page directory base pointer.

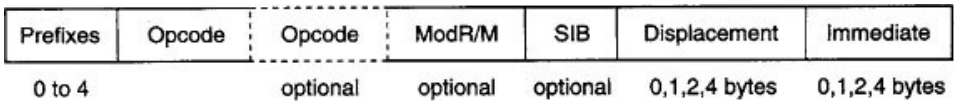


Figure A.26 General Format for IA-32 Instructions.

Instruction Set Summary

The IA-32 ISA has variable-length instructions, from one to fifteen bytes, although most of the commonly used instructions are relatively short. Instruction operands, including instructions that perform common arithmetic, logical, or shift operations, may come from either registers or memory. This is in contrast to RISC instruction sets, where the only instructions that take operands from memory are the load and store instructions.

Figure A.26 illustrates the general form of an IA-32 instruction. It consists of from zero to four prefix bytes. The prefix bytes indicate various special cases, for example, if there is repetition for string instructions and/or if there are overrides for addressing segments, address sizes, and operand sizes. Following the prefix byte(s) (if any) is an opcode byte, which may be followed by a second opcode byte, depending on the value of the first. Next comes an optional addressing-form specifier, ModR/M. It is present only for certain opcodes and generally indicates an addressing mode and register. The SIB byte is present only for certain ModR/M encodings and indicates a base register, an index register, and a scale factor for the indexing. The variable-length displacement field is present only when certain

addressing modes are specified. The last field is a variable-length immediate operand, present only if required by the opcode. When needed, a segment register can be specified with some of the opcode bits or by the ModR/M byte.

The IA-32 ISA divides the user instruction set into three major categories: integer instructions, floating-point instructions, and MMX (multimedia instructions). In this brief overview and in examples used in this book, we restrict ourselves primarily to the integer instructions. Within the category of integer instructions are jumps (both conditional and unconditional), load/store instructions, and a wide variety of ALU instructions. However, as we noted earlier, the ALU instructions can specify one or more memory operands, so, in effect, many of them also perform load/store operations.

Jump instructions include all control transfer instructions, regardless of whether they are conditional or unconditional. The conditional jumps test condition code bits that are held in the EFLAGS register. These bits are set as a side effect of many instructions.

jmp	reg	Jump indirect to the memory location held in reg.
jz	target	Jump conditionally to target if the zero-condition code bit in the EFLAGS register is set to 1.
jmp	target	Jump unconditionally to target.

Procedure calls and returns are performed using instructions that push and pop the return address to/from the stack, pointed to by esp.

call	target	Push the program counter of the next instruction onto the stack and jump to the target address.
ret		Pop the return address from the stack and jump.

Load and store operations, when not combined with arithmetic or logical operations, are performed by move instructions.

mov	reg1, disp(reg2)	Load register reg1 with the contents of memory addressed by (reg2) + disp. The disp field is treated as a signed quantity.
mov	disp(reg1), reg2	Store register reg2 into memory addressed by (reg1) + disp. The disp field is treated as a signed quantity.

Most of the ALU operations can take operands from a combination of registers and memory locations. Some representative examples follow.

addl	reg1, disp(reg2)	Add the contents of reg1 and the memory operand at address reg2 + disp. The result is placed in reg1. Condition codes in the EFLAGS register are set according to the result.
sub	reg1, immed	Subtract the contents of the immediate value immed from reg1 and place the result in reg1. Condition codes in the EFLAGS register are set according to the result.
xorl	reg1, reg2	Exclusive-OR the contents of reg1 and reg2 and place the result in reg1. Condition codes in the EFLAGS register are set according to the result.

---

1. Complex-instruction-set computer.

2. Reduced-instruction-set computer.

3. This distinction between branch and jump instructions is not universal. Some ISAs use only one of the two terms, and others interchange their meanings.

Previous chapter
< Chapter 10: Emerging Applications

Next chapter
References >

References

Adair, R., R. U. Bayles, L. W. Comeau, and R. J. Creasy. 1966. A Virtual Machine System for the 360/40, *Cambridge Scientific Center Report No. G320–2007* (May).

Adve, V., C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. 2003. LLVA: A Low-level Virtual Instruction Set Architecture, *Proc. 36th Int. Symp. on Microarchitecture* (December).

Aho, A. V., R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA.

Allcock, B., J. Bester, J. Breshanan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. 2001. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing, *Proc. 18th Symposium on Mass Storage Systems and Technologies* (April).

Altman, E., K. Ebcioglu, M. Gschwind, and S. Sathaye. 2002. Methods and Apparatus for Reordering and Renaming Memory References in a Multiprocessor Computer System, US patent 6,349,361 (February).

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr. 1964. Architecture of the IBM System 360, *IBM Journal of Research and Development* (April), pp. 87–101.

Anderson, D. P., J. Cobb, E. Korpella, M. Lebofsky, and D. Werthimer. 2002. SETI@home: An Experiment in Public-Resource Computing, *Communications of the ACM* (November), pp. 56–61.

Appel, A. W. 1991. Garbage Collection, *Topics in Advanced Language Implementations,* P. Lee (Ed.), MIT Press, Cambridge, MA, pp. 89–100.

ARM. 2002, ARM 9EJ-S Technical Reference Manual, Ref. DDI0222B.

Arnold, M. and B. G. Ryder. 2001. A Framework for Reducing the Cost of Instrumented Code, *Proc. Conf. ACM SIGPLAN '01 on Programming Language Design and Implementation* (May), pp. 168–179.

Arnold, M., M. Hind, and B. G. Ryder. 2002. Online Feedback-Directed Optimization of Java, *Proc. Conf. on Object Oriented Programming Systems, Languages, and Applications* (November), pp. 111–129.

Arnold, M., S. Fink, D. Grove, M. Hind, and P. F. Sweeney. 2000. Adaptive Optimization in the Jalapeno JVM, *Proc. Conf. on Object Oriented Programming Systems, Languages, and Applications* (October), pp. 47–65.

Arnold, M., S. Fink, D. Grove, M. Hind, and P. F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines, *Proceedings of the IEEE* (February).

Artigas, P. V., M. Gupta, S. P. Midkiff, and J. E. Moreira. 2000. Automatic Loop Transformations and Parallelization for Java, *Proc. Int. Conf. on Supercomputing* (May), pp. 1–10.

Attanasio, C. R., D. F. Bacon, A. Cocchi, and S. Smith. 2001. A Comparative Evaluation of Parallel Garbage Collector Implementations, *Proc. 14th Workshop on Languages and Compilers for Parallel Computing* (August), pp. 177–192.

Aycock, J. 2003. A Brief History of Just-in-Time, *ACM Comp. Surveys* (June), pp. 97–113.

Ayers, A., R. Schooler, and R. Gottlieb. 1997. Aggressive Inlining, *Proc. ACM SIGPLAN Conf. on Programming Language Design and*

Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (May), pp. 134–145.

Bala, V., E. Duesterwald, and S. Banerjia. 1999. Transparent Dynamic Optimization: The Design and Implementation of Dynamo, *Hewlett Packard Laboratories Technical Report HPL-1999-78* (June).

Bala, V., E. Duesterwald, and S. Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System, *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (June), pp. 1–12.

Ball, T. and J. R. Larus. 1994. Optimally Profiling and Tracing Programs, *ACM Transactions on Programming Languages and Systems* (July), pp. 1319–1360.

Ball, T. and J. R. Larus. 1996. Efficient Path Profiling, *Proc. 29th Int. Symp. on Microarchitecture* (November), pp. 46–57.

Ball, T., P. Mataga, and M. Sagiv. 1998. Edge Profiling Versus Path Profiling: The Showdown, *Proc. 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (January), pp. 134–148.

Banerjia, S., W. A. Havanki, and T. M. Conte. 1997. Treegion Scheduling for Highly Parallel Processors, *European Conference on Parallel Processing* (August), pp. 1074–1078.

Banning, J., P. H. Anvin, B. Gribstad, D. Keppel, A. Klaiber, and P. Serris. 2002. Fine Grain Translation Discrimination, US Patent 6,363,336 (March).

Baraz, L, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. 2003. IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems, *Proc. 36th Annual IEEE/ACM Int. Symp. on Microarchitecture* (December), pp. 191–204.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. 2003. Xen and the Art of Virtualization, *Proc. 19th ACM Symp. on Operating System Principles* (October), pp. 164–177.

Bell, J. R. 1973. Threaded Code, *Communications of the ACM* (June), pp. 370–372.

Berndl, M. and L. Hendren. 2003. Dynamic Profiling and Trace Cache Generation, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 276–285.

Bertsis, V. 1980. Security and Protection of Data in the IBM System/38, *Proc. 7th Int. Symp. on Computer Architecture* (June), pp. 245–252.

Boehm, H. and M. Weiser. 1988. Garbage Collection in an Uncooperative Environment, *Software—Practice and Experience,* pp. 807–820.

Borden, T. L., J. P. Hennessy, and J. W. Rymarczyk. 1989. Multiple Operating Systems on One Processor Complex, *IBM Systems Journal* (January), pp. 104–123

Boutcher, D. 2001. The Linux Kernel on iSeries, *Proc. 2001 Ottawa Linux Symposium* (July), **http://lwn.net/2001/features/OLS/pdf/pdf/iseries.pdf**.

Bovet, D. P. and M. Cesati. 2001. *Understanding the Linux Kernel, O'Reilly,* Sebastopol, CA.

Bowles, K. L. 1980. *Beginner's Guide for the UCSD Pascal System,* McGraw-Hill, New York.

Box, D. 2002. *Essential .NET, Volume 1: The Common Language*

Beck, B. 2002. *Essential .NET, Volume 1: The Common Language Runtime,* Addison-Wesley, Reading, MA.

Bressoud, T. C. and F. B. Schneider. 1996. Hypervisor-Based Fault Tolerance, *ACM Transactions on Computer Systems* (February), pp. 80–107.

Bruening, D. 2004. Efficient, Transparent, and Comprehensive Runtime Code Manipulation, Ph.D. dissertation (September), MIT, Cambridge, MA.

Bruening, D., E. Duesterwald, and S. Amarasinghe. 2001. Design and Implementation of a Dynamic Optimization Framework for Windows, *Proc. 4th Workshop on Feedback-Directed and Dynamic Optimization* (December).

Bruening, D., T. Garnett, and S. Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 265–275.

Brunner, R. A. 1991. *VAX Architecture Reference Manual,* 2nd ed., Digital Press, Bedford, MA.

Bugnion, E., S. Devine, K. Govil, and M. Rosenblum. 1997. Disco: Running Commodity Operating Systems on Scalable Multiprocessors, *ACM Transactions on Computing Systems* (November), pp. 412–447.

Carr, S. and K. Kennedy. 1994. Scalar Replacement in the Presence of Conditional Control Flow, *Software—Practice and Experience* (January), pp. 51–77.

Chang, P. P., S. A. Mahlke, and W. W. Hwu. 1991. Using Profile Information to Assist Classic Code Optimizations, *Software—Practice & Experience* (December), table of contents, pp. 1301–1321.

Chang, P.P., S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. 1992. Profile-Guided Automatic Inline Expansion for C Programs, *Software—Practice and Experience* (May), pp. 349–369.

Chambers, C. and D. Ungar. 1991. Making Pure Object-Oriented Languages Practical, *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications* (October), pp. 1–15.

Charlu, D. 1999. HP Hyperplex Clustering Technology, *1st Int. Workshop on Cluster Computing* (December).

Chen, P. M. and B. D. Noble. 2001. When Virtual Is Better Than Real, *Proc. 8th IEEE Workshop on Hot Topics on Operating Systems* (May), pp. 133–138.

Chen, J. B. and B. D. D. Leupen. 1997. Improving Instruction Locality with Just-In-Time Code Layout, *Proc. the USENIX Windows NT Workshop* (August).

Chen, W.-K., S. Lerner, R. Chaiken, and D. M. Gillies. 2000. Mojo: A Dynamic Optimization System, *Proc. 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization* (December).

Chernoff, A., M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. 1998. FX!32: A Profile-Directed Binary Translator, *IEEE Micro* (March), pp. 56–64.

Chien, A., B. Calder, S. Elbert, and K. Bhatia. 2003. Entropia: Architecture and Performance of an Enterprise Desktop Grid System, *Journal of Parallel Distributed Computing* (May), pp. 597–610.

Chilimbi, T. M., B. Davidson, and J. R. Larus. 1999. Cache-Conscious Structure Definition, *Proc. Conf. on Programming Language Design and Implementation* (May), pp. 13–24.

Choi, J.-D., M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff.

1999. Escape Analysis for Java, *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 1–19.

Choi, Y., A. Knies, G. Vedaraman, J. Williamson, and I. Esmer. 2002. Design and Experience: Using the Intel Itanium-2 Processor Performance Monitoring Unit to Implement Feedback Optimizations, *2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers* (November).

Chou, Y. and J. Shen. 2000. Instruction Path Coprocessors, *Proc. 27th Int. Symp. on Computer Architecture* (June), pp. 270–281.

Cifuentes, C. and M. V. Emmerik. 2000. UQBT: Adaptable Binary Translation at Low Cost, *IEEE Computer* (March), pp. 60–66.

Cifuentes, C., B. Lewis, and D. Ung. 2002. Walkabout—A Retargetable Dynamic Binary Translation Framework, *Proc. Workshop on Binary Translation* (September).

Cmelik, B. and D. Keppel. 1994. Shade: A Fast Instruction-Set Simulator for Execution Profiling, *ACM Sigmetrics* (May), pp. 128–137.

Cmelik, R. F. and D. Keppel. 1996. Shade: A Fast Instruction-Set Simulator for Execution Profiling, *Technical Report UWCSE 93-06-06,* University of Washington, Seattle (June).

Cohen, D. 1981. On Holy Wars and a Plea for Peace, *IEEE Computer* (October), pp. 48–54.

Cohn, R. S., D. W. Goodwin, P. G. Lowney, and N. Rubin. 1997. Spike: An Optimizer for Alpha/NT Executables, *USENIX Windows NT Workshop* (August), pp. 17–23.

Collins, G. E. 1960. A Method for Overlapping and Erasure of Lists,

Communications of the ACM (December), pp. 655–657.

Comfort, W. T. 1964. Multiword List Items, *Communications of the ACM* (June), pp. 357–362.

Conte, T. M. and S. Sathaye. 1995. Dynamic Rescheduling: A Technique for Object Code Compatibility, *Proc. 28th Int. Symp. on Microarchitecture* (November), pp. 208–217.

Conte, T. M., K. N. Menezes, and M. A. Hirsch. 1996. Accurate and Practical Profile-Driven Compilation Using the Profile Buffer, *Proc. 29th Int. Symp. on Microarchitecture* (November), pp. 36–45.

Conte, T., S. Sathaye, and S. Banerjia. 1996. A Persistent Rescheduled-Page Cache for Low Overhead Object Code Compatibility in VLIW Architectures, *Proc. 29th Int. Symp. on Microarchitecture* (December), pp. 4–13.

Cooper, K. and L. Torczon. 2003. *Engineering a Compiler,* Morgan-Kaufmann, San Francisco.

Creasy, R. J. 1981. The Origin of the VM/370 Time-Sharing System, *IBM Journal of Research and Development* (September), pp. 483–490.

Culler, D. E. and J. P. Singh. 1999. *Parallel Computer Architecture: A Hardware/Software Approach,* Morgan-Kaufmann, San Francisco.

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems* (October), pp. 451–490.

Czajkowski, K., I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Teucke. 1998. A Resource Management Architecture for Meta-computing Systems, *Proc. 4th Workshop on Job Scheduling Strategies for Parallel Processing* (March).

Dean, J., J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. 1997. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors, *Proc. 30th Int. Symp. on Microarchitecture* (November), pp. 292–302.

Deaver, D., R. Gorton, and N. Rubin. 1999. Wiggins/Redstone: An Online Program Specializer, *Proc. 11th HotChips Symposium* (June).

Debaere, E. H. and J. M. Van Campenhout. 1990. *Interpretation and Instruction Path Coprocessing*, MIT Press, Cambridge, MA.

Dehnert, J. C., B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. 2003. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 15–24.

Desoli, G., N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. 2002. DELI: A New Run-Time Control Point, *Proc. 35th Int. Symp. on Microarchitecture* (November), pp. 257–268.

Deutsch, P. and A. M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System, *Proc. 11th ACM Symp. on Principles of Programming Languages* (January), pp. 297–302.

Dewar, R. B. K. 1975. Indirect Threaded Code, *Communications of the ACM* (June), pp. 330–331.

DHCP. Dynamic Host Configuration Protocol, **http://www.dhcp.org**.

Diffie, W. and M. E. Hellman. 1976. New Directions in Cryptography, *IEEE Trans. on Information Theory* (November), pp. 644–654.

Doran, R. W. 1988. Amdahl Multiple Domain Architecture, *IEEE Computer* (October), pp. 20–28.

Computer (October), pp. 20–28.

Dorward, D., R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. 1997. Inferno, *Proc. IEEE Compcon '97.*

Drayton, P., B. Albahari, and T. Neward. 2002. *C# in a Nutshell,* 1st ed., O'Reilly, Sebastopol, CA.

Duesterwald, E. and V. Bala. 2000. Software Profiling for Hot Path Prediction: Less Is More, *Proc. 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (November), pp. 202–211.

Dunlap, G. W., S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay, *Proc. 5th Symp. on Operating Systems Design and Implementation* (December), pp. 211–224.

Ebcioglu, K., E. R. Altman, M. Gschwind, and S. Sathaye. 2001. Dynamic Binary Translation and Optimization, *IEEE Transactions on Computers* (June), pp. 529–548.

Engebretson, D., M. Corrigan, and P. Bergner. 2001. PowerPC 64-bit Kernel Internals, *Proc. 2001 Ottawa Linux Symposium* (July), **http://lwn.net/2001/features/OLS/pdf/pdf/ppc64.pdf**.

Ertl, M. A. and D. Gregg. 2001. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures, *Europar 2001,* pp. 403–412.

Ertl, M. A. and D. Gregg. 2003. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters, *Proc. Conf. on Programming Language Design and Implementation* (May) pp. 278–288.

Fahs, B., S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel,

and S. S. Lumetta. 2001. Performance Characterization of a Hardware Mechanism for Dynamic Optimization, *Proc. 34th Int. Symp. on Microarchitecture* (December), pp. 16–27.

Figueiredo, R., P. Dinda, and J. Fortes. 2003. A Case for Grid Computing on Virtual Machines, *Proc. 23rd Int. Conf. on Distributed Computing Systems* (May).

Fink, S. J. and F. Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 241–252.

Fisher, J. A. 1981. Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Transactions on Computers* (July), pp. 478–490.

Flanagan, D. 1999. *Java in a Nutshell,* 3rd ed., O'Reilly, Sebastopol, CA.

FLEX. FLEX-ES: The New Mainframe, Fundamental Software Inc., **http://www.funsoft.com**.

Foster, I. and N. T. Karonis. 1998. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems, *Proc. Intl. Conf. ACM/IEEE on Supercomputing* (November).

Foster, I. and C. Kesselman (Eds.). 1998. *The Grid: Blueprint for a Future Computing Infrastructure,* Morgan Kaufmann, San Francisco.

Foster, I. and C. Kesselman (Eds.). 2004. *The Grid 2: Blueprint for a Future Computing Infrastructure,* Morgan Kaufmann, San Francisco.

Foster, I., C. Kesselman, and S. Tuecke. 2001. The Anatomy of a Grid: Enabling Virtual Scalable Organizations, *Int. Journal of High Performance Computing Applications,* pp. 200–222.

Fujitsu. 2003. The Next-Generation Engine for Consolidation, **http://www.computers.us.fujitsu.com/www/content/aboutus/analysts/data/** FujitsuGroup_ConsolidationEngine.pdf.

Gallagher, D., W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. 1994. Dynamic Memory Disambiguation Using the Memory Conflict Buffer, *Proc. 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (October), pp. 183–193.

Garfinkel, T. and M. Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Symp. of the Internet Society's 2003 on Network and Distributed System Security* (February).

Garfinkel, T., B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. 2003. Terra: A Virtual Machine-Based Platform for Trusted Computing, *Proc. 19th Symp. on Operating System Principles* (October).

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Int. Symp. on Computer Architecture* (May), pp. 15–26.

Gill, S. 1951. The Diagnosis of Mistakes in Programmes on the *EDSAC, Proc. of the Royal Society Series A, Mathematical and Physical Sciences* (May), pp. 538–554.

Globus. The Globus Alliance, **http://www.globus.org**.

Goldberg, R. P. 1972. Architectural Principles for Virtual Computer Systems, Ph.D. dissertation, Harvard University, Cambridge, MA.

Goldberg, R. P. 1974. Survey of Virtual Machine Research, *IEEE Computer* (June), pp. 34–45.

Gong, L., G. Ellison, and M. Dageforde. 2003. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation,* 2nd ed., Addison-Wesley, Reading, MA.

Gosling, J., B. Joy, and G. Steele. 1996. *The Java Language Specification,* Addison Wesley, Reading, MA.

Govil, K., D. Teodosiu, Y. Huang, and M. Rosenblum. 1999. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors, *Proc. 17th ACM Symp. on Operating System Principles* (December), NC, pp. 154–169.

Grant, B., M. Philipose, M. Mock, S. J. Eggers, and C. Chambers. 1999. An Evaluation of Run-Time Optimizations, *Proc. Conf. on Programming Language Design and Implementation* (May), pp. 293–304.

Gschwind, M. 1998a. Method and Apparatus for Determining Branch Addresses in Programs Generated by Binary Translation, *IBM Research Disclosures YOR819980334* (July).

Gschwind, M. 1998b. Method and Apparatus for Rapid Return Address Computation in Binary Translation, *IBM Research Disclosures YOR819980410* (September).

Gschwind, M. and E. R. Altaian. 2000. Optimization and Precise Exceptions in Dynamic Compilation, *Proc. Workshop on Binary Translation* (October).

Gschwind, M., K. Ebcioglu, E. R. Altman, and S. Sathaye. 2000. Binary Translation and Architecture Convergence Issues for IBM System/390, *Proc. Int. Conf. on Supercomputing* (May), pp. 336–347.

Gum, P. H. 1983. System/370 Extended Architecture: Facilities for Virtual Machines, *IBM Journal of Research and Development*

(November), pp. 530–544.

Gupta, M., J.-D. Choi, and M. Hind. 2000. Optimizing Java Programs in the Presence of Exceptions, *14th European Conference on Object-Oriented Programming* (June), pp. 422–446.

Halfhill, T. R. 2000. Transmeta Breaks x86 Low-Power Barrier, *Microprocessor Report* (February 14), pp. 9–18.

Hall, J. S. and P. T. Robinson. 1990. Virtualizing the VAX Architecture, *Proc. USENIX1990 Summer Conference* (June), pp. 380–389.

Hansen, G. J. 1974. Adaptive Systems for the Dynamic Run-Time Optimization of Programs, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA.

Hazelwood, K. and J. E. Smith. 2004. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems, *Proc. Second Annual IEEE/ACM Int. Symp. on Code Generation and Optimization* (March), pp. 89–99.

Heil, T. and J. E. Smith. 2000. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines, *Proc. 33rd Int. Symp. on Microarchitecture* (December), pp. 281–290.

Hennessy, J. and D. Patterson. 2002. *Computer Architecture: A Quantitative Approach,* 3rd ed., Morgan Kaufmann, San Francisco.

Hennessy, J., M. Heinrich, and A. Gupta. 1999. Cache-Coherent Distributed Shared Memory: Perspectives on its Development and Future Challenges, *Proceedings of the IEEE* (March), pp. 418–429.

Hewlett-Packard. 2000. HP Partitioning Continuum, *Technical Positioning White Paper* (June), **http://h30081.www3.hp.com/products/wlm/docs/HPPartitioningContinuum.pdf**.

Hinton, G., D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. 2001. The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal* (Ql).

Hohensee, P., M. Myszewski, and D. Reese. 1996. Wabi CPU Emulation, *Proc. 8th HotChips Symposium* (August), pp. 47–65.

Hölzle, U. and D. Ungar. 1996. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages, *ACM Transactions on Programming Languages and Systems* (July), pp. 355–400.

Hölzle, U., C. Chambers, and D. Ungar. 1991. Optimizing Dynamically Typed Object-Oriented Languages with Polymorphic Inline Caches, *5th European Conference on Object-Oriented Programming* (July), pp. 21–38.

Hölzle, U., C. Chambers, andD. Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization, *Proc. Conf. on Programming Language Design and Implementation* (July), pp. 32–43.

Hookway, R. J. and M. A. Herdeg. 1997. Digital FX!32: Combining Emulation and Binary Translation, *Digital Technical Journal* (January), pp. 3–17.

Horspool, R. N. and N. Marovac. 1980. An Approach to the Problem of Detranslation of Computer Programs, *Computer Journal* (August), pp. 223–229.

Höxer, H.-J., K. Buchacker, and V. Sieh. 2002. UMLinux—A Tool for Testing a Linux System's Fault Tolerance, *Proc. LinuxTag* (June).

Hsu P.-T. and E. Davidson. 1986. Highly Concurrent Scalar Processing, *Proc. 13th Int. Symp. on Computer Architecture* (June), pp. 386–395.

Hunt, G., and D. Brubacher. 1999. Detours: Binary Interception of Win32 Functions, *Proc. 3rd USENIX Windows NT Symposium* (July), pp. 135–143.

Hwu W. W., S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. 1993. The Superblock: An Effective Technique for VLIW and Superscalar Compilation, *Journal of Supercomputing* (May) pp. 229–248.

IBM. 1994. *The PowerPC Architecture,* Morgan Kaufmann, San Francisco.

IBM. 2001. *Autonomic Computing: IBM's Perspective on the State of Information Technology,* Armonk, NY (October).

Intel. 1999. *Intel(R) Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual,* **http://www.intel.com/design/intarch/manuals/243191.htm**.

Intel. 2005. *Intel Vanderpool Technology for IA-32 Processors (VT-x) Preliminary Specification,* Order Number C97063-001, Santa Clara, CA (January).

Jacob, B. and T. Mudge. 1998. Virtual Memory: Issues of Implementation, *IEEE Computer* (June), pp. 33–43.

Jann, J., L. M. Browning, and R. S. Burugula. 2003. Dynamic Reconfiguration: Basic Building Blocks for Autonomic Computing on IBM pSeries Servers, *IBM Systems Journal* (January), pp. 29–37.

Jones, R. 1996. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, New York.

Kaeli, D. and P. G. Emma. 1991. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns, *Proc. 18th Int.*

Symp. on Computer Architecture (June), pp. 34–42.

Kane, G. 1996. *PA-RISC Architecture,* Prentice Hall, New York.

Keefe, D. D. 1968. Hierarchical Control Program for System Evaluation, *IBM Systems Journal,* pp. 123–133.

Kelly, E. J., R. F. Cmelik, and M. J. Wing. 1998. Memory Controller for a Microprocessor for Detecting a Failure of Speculation on the Physical Nature of a Component Being Addressed, US patent 5,832,205 (November).

Keltcher, C. N., K. J. McGrath, A. Ahmed, and P. Conway. 2003. The AMD Opteron Processor for Multiprocessor Servers, *IEEE Micro* (March/April), pp. 66–76.

Kim, H.-S. and J. E. Smith. 2003. Dynamic Binary Translation for Accumulator-Oriented Architectures, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 25–35.

Kiriansky, V., D. Bruening, and S. Amarasinghe. 2002. Secure Execution via Program Shepherding, *Proc. 11th USENIX Security Symposium* (August).

Kistler, T. and M. Franz. 2000. Automated Data-Member Layout of Heap Objects to Improve Memory-Hierarchy Performance, *ACM Transactions on Programming Languages and Systems* (May), pp. 490–505.

Kistler, T. and M. Franz. 2001. Continuous Program Optimization: Design and Evaluation, *IEEE Transactions on Computers* (June), pp. 549–566.

Klaiber, A. 2000. The Technology Behind Crusoe Processors, *Transmeta Technical Brief.*

Klint, P. 1981. Interpretation Techniques, *Software Practice and Experience* (September), pp. 963–973.

Kogge, P. M. 1982. An Architectural Trail to Threaded-Code Systems, *IEEE Computer* (March), pp. 22–34.

Kozuch M. and M. Satyanarayanan. 2002. Internet Suspend/Resume, *Proc. 4th IEEE Workshop on Mobile Systems and Applications* (June).

Lam, M. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proc. ACM Conf. on Programming Languages Design and Implementation* (June), pp. 318–328.

Lamport, L. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers* (September), pp. 690–691.

Larus, J. 1991. *SPIM S20: A MIPS R2000 Simulator*, University of Wisconsin—Madison Technical Report, **http://www.cs.wisc.edu/~larus/spim.html**.

Larus, J. R. and E. Schnarr. 1995. EEL: Machine-Independent Executable Editing, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (June), pp. 291–300.

Lawton, K. The Bochs IA-32 Emulator Project, **http://bochs.sourceforge.net**.

Le, B. C. 1998. An Out-of-Order Execution Technique for Runtime Binary Translators, *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* pp. 151–158.

Lidin, S. 2002. *Inside Microsoft .NET IL Assembler*, Microsoft Press, Redmond, WA.

Lindholm, T. and F. Yellin. 1999. *The Java Virtual Machine Specification,* 2nd ed., Addison-Wesley, Reading, MA.

Lowney, P. G., S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. Ruttenberg. 1993. The Multiflow Trace Scheduling Compiler, *Journal of Supercomputing* (May), pp. 51–142.

Lu, J., H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. 2003. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System, *Proc. 36th IEEE/ACM Int. Symp. on Microarchitecture* (December), pp. 180–190.

Lu, J., H. Chen, P.-C. Yew, and W.-C. Hsu. 2004. Design and Implementation of a Lightweight Dynamic Optimization System, *Journal of Instruction-Level Parallelism,* Vol. 6, pp. 1–24.

Lucco, S., O. Sharp, and R. Wahbe. 1995. Omniware: A Universal Substrate for Web Programming, *World Wide Web Journal* (winter), pp. 359–368.

MacKinnon, R. A. 1979. The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines, *IBM Systems Journal,* pp. 18–46.

Madnick, S. E. and J. J. Donovan. 1974. *Operating Systems,* McGraw-Hill, New York.

Magnusson, P. and D. Samuelsson. 1994. A Compact Intermediate Format for SIMICS, *Tech. Report R94:17,* Swedish Institute of Computer Science (September).

Mahlke, S. A., D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. 1992. Effective Compiler Support for Predicated Execution Using the Hyperblock, *Proc. 25th Int. Symp. on Microarchitecture* (November), pp. 45–54.

May, C. 1987. Mimic: A Fast System/370 Simulator, *Conf. on Programming Language Design and Implementation,* archive, pp. 1–13.

McGhan, H. and M. O'Connor. 1998. Picojava: A Direct Execution Engine for Java Bytecode, *IEEE Computer* (October), pp. 22–30.

McGraw, B. and E. W. Felten, 1999. *Securing Java: Getting Down to Business with Mobile Code,* 2nd ed., Wiley, New York.

Meloan, S. 1999. The Java HotSpot Performance Engine: An In-Depth Look (June), **http://java.sun.com/developer/technicalArticles/Networking/HotSpot**.

Merten, M. C, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu. 2000. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hotspots, *Proc. 27th Int. Symp. on Computer Architecture* (June), pp. 59–70.

Merten, M. C, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. 2001. An Architectural Framework for Run-Time Optimization, *IEEE Transactions on Computers* (June), pp. 567–589.

Meyer, R. A. and L. H. Seawright. 1970. A Virtual Machine Time-Sharing System, *IBM Systems Journal,* pp. 199–218.

Myers, G. J. 1982. *Advances in Computer Architecture,* 2nd ed., Wiley, New York.

Nair, R. and M. E. Hopkins. 1997. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups, *Proc. 24th Int. Symp. on Computer Architecture* (June), pp. 13–25.

Nethercote, N., and J. Seward. 2003. Valgrind: A Program

Supervision Framework, *Proc. 3rd Workshop on Runtime Verification* (July).

Nick, J. M., B. B. Moore, J.-Y. Chung, and N. S. Bowen. 1997. S/390 Cluster Technology: Parallel Sysplex, *IBM Systems Journal,* pp. 172–201.

NIST. 2002. FIPS 180–2, Secure Hash Standard (SHS) (August).

Nori, K. V., U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi. 1975. *The Pascal P-Compiler: Implementation Notes* (rev. ed.), Institut fur Informatik ETH, Zurich, Tech. Rep. 10.

Nystrom, E., R. D. Barnes, M. C. Merten, and W. W. Hwu. 2001. Code Reordering and Speculation Support for Dynamic Optimization Systems, *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques* (September), pp. 163–174.

OGSA. Open Grid Services Architecture Working Group, **https://forge.gridforum.org/projects/ogsa-wg**.

OpenSSH. OpenSSH, **http://www.openssh.com**.

Osisek, D. L., K. M. Jackson, and P. H. Gum. 1991. ESA/390 Interpretive Execution Architecture, Foundation for VM/ESA, *IBM Systems Journal,* pp. 34–51.

Pacheco, P. 1996. *Parallel Programming with MPI,* Morgan-Kaufmann, San Francisco.

Paleczny, M., C. Vick, and C. Click. 2001. The Java Hotspot Server Compiler, *USENIX Java Virtual Machine Research and Technology Symp.* (April), pp. 1–12.

Parmalee, R. P., T. I. Peterson, C. C. Tillman, and D. J. Hatfield. 1972. Virtual Storage and Virtual Machine Concepts, *IBM Systems Journal,*

pp. 99–219.

Patel, S. J., S. S. Lumetta. 2001. rePLay: A Hardware Framework for Dynamic Optimization, *IEEE Transactions on Computers* (June), pp. 590–608.

Perkins, C. E. 1998. Mobile Networking Through Mobile IP, *IEEE Internet Computing* (January), pp. 58–69.

Pettis, K. and R. C. Hansen. 1990. Profile Guided Code Positioning, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (June), pp. 16–27.

Popek, G. J. and R. P. Goldberg. 1974. Formal Requirements for Virtualizable Third-Generation Architectures, *Communications of the ACM* (July), pp. 412–421.

Richardson T., Q. Stafford-Fraser, K. R. Wood, and A. Hopper. 1998. Virtual Network Computing, *IEEE Internet Computing* (January), pp. 33–38.

Ridge, D., D. Becker, P. Merkey, and T. Sterling. 1997. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs, *Proc. IEEE Aerospace Conference* (February), pp. 79–91.

Robin, J. S. and C. E. Irvine. 2000. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, *Proc. 9th USENIX Security Symposium,* Denver (August).

Romer, T. H., D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. 1996. The Structure and Performance of Interpreters, *Proceedings 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (September), pp. 150–159.

Ronsse, M. and K. De Bosschere. 2000. JiTI: A Robust Just-in-Time

Instrumentation Technique, *Proc. Workshop on Binary Translation* (October).

Rosenblum, M. 2000. VMware's Virtual Platform Technology, *Multi-University/Research Laboratory Seminar Series,* **http://murl.microsoft.com/LectureDetails.asp?530** (December).

Rotenberg, E., S. Bennett, and J. E. Smith. 1996. Trace Cache: A Low-Latency Approach to High-Bandwidth Instruction Fetching, *Proc. 29th Int. Symp. on Microarchitecture* (December), pp. 24–35.

Sapuntzakis, C. P., R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. 2002. Optimizing the Migration of Virtual Computers, *Proc. 5th Symp. on Operating Systems Design and Implementation* (December).

Sathaye, S., P. Ledak, J. LeBlanc, S. Kosonocky, M. Gschwind, J. Fritts, Z. Filan, A. Bright, D. Appenzeller, E. Altman, and C. Agricola. 1999. BOA: Targeting Multi-Gigahertz with Binary Translation, *Proc. Workshop on Binary Translation* (October).

Scheifler, R. W. 1977. An Analysis of Inline Substitution for a Structured Programming Language, *Communications of the ACM* (September), pp. 647–654.

Scott, K., N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Sofia. 2003. Retargetable and Reconfigurable Software Dynamic Translation, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 36–47.

Shannon, B., M. Hapner, V. Matena, E. Pelegri-Llopart, J. Davidson, and L. Cable. 2000. *Java 2 Platform, Enterprise Edition: Platform and Component Specifications,* Addison-Wesley, Reading, MA.

Sites, R. L., A. Chernoff, M. B. Kerk, M. P. Marks, and S. G. Robinson.

1993. Binary Translation, *Communications of the ACM* (February), pp. 69–81.

Smith, M. D., M. S. Lam, and M. A. Horowitz. 1990. Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. 17th Int. Symp. on Computer Architecture* (June), pp. 344–354.

Soltis, F. G. 1996. *Inside the AS/400,* Duke Press, Loveland, CO.

SPEC. Standard Performance Evaluation Corporation, **www.spec.org**.

Stichnoth, J., G.-Y. Lueh, and M. Cuerniak. 1999. Support for Garbage Collection at Every Instruction in a Java Compiler, *Proc. Conf. on Programming Language Design and Implementation* (May), pp. 118–127.

Suganuma, T., T. Yasue, and T. Nakatani. 2002. An Empirical Study of Method In-Lining for a Java Just-in-Time Compiler, *Usenix Java Virtual Machine Research and Technology Symp.* (August), pp. 91–104.

Suganuma, T., T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. 2000. Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal* (February), pp. 175–193.

Suganuma, T., T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. 2001. A Dynamic Optimization Framework for a Java Just-in-Time Compiler, *Proc. Conf. on Object Oriented Programming, Systems, Languages and Applications* (November), pp. 180–195.

Sugerman, J., G. Venkitachalam, and B.-H. Lim. 2001. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *Proc. USENIX Annual Technical Conference* (June), pp. 1–14.

Sun Microsystems. 1999. Sun Enterprise 10000 Server: Dynamic Systems Domains, *Technical White Paper* **http://www.sun.com/data-center/docs/domainswp.pdf**.

Sunderaraj, A. I. and P. A. Dinda. 2004. Towards Virtual Networks for Virtual Machine Grid Computing, *Proc. 3rd USENIX Virtual Machine Technology Symposium* (May).

Tabatabai, A., M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. 1998. Fast Effective Code Generation in a Just-In-Time Java Compiler, *Proc. Conf. on Programming Language Design and Implementation* (June), pp. 280–290.

Tamches, A. and B. P. Miller. 1999. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels, *Proc. 3rd Symp. on Operating Systems Design and Implementation* (February), pp. 117–130.

Traut, E. 1997. Building the Virtual PC, *Byte* (November), pp. 51–52.

Varian, M. 1997. VM and the VM Community: Past, Present, and Future, *SHARE 89* (August).

Venners, B. 1998. *Inside the Java Virtual Machine,* McGraw-Hill, New York.

VirtualCenter. *VMware VirtualCenter Users Manual, version 1.2,* VMware Inc., Palo Alto, CA, **http://www.vmware.com/pdf/VC_Users_Manual_ll.pdf**.

VMware. 2001. *VMware GSX Server User Manual, version 1.0,* VMware Inc., Palo Alto, CA.

VMware. 2000. VMware Virtual Platform Technical White Paper, **http://ftp.hit.edu.cn/linux/Datas/Docs/webmirror/**

www.vmware.com/virtualplatform.html.

Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham. 1993. Efficient Software-Based Fault Isolation, *ACM SIGOPS Operating Systems Review* (December), pp. 203–216.

Wallach, D. and E. Felten. 1998. Understanding Java Stack Inspection, *Proc. IEEE Symp. on Security and Privacy* (May), pp. 52–63.

Whaley, J. 2001. Partial Method Compilation Using Dynamic Profile Information, *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications* (October), pp. 166–179.

Whitaker, A., M. Shaw, and S. D. Gribble. 2002. Denali: Lightweight Virtual Machines for Distributed and Networked Applications, *Technical Report 02-02-01,* University of Washington, Seattle.

Wilson, P. R. 1992. Uniprocessor Garbage Collection Techniques, *Proc. Int. Workshop on Memory Management* (September), pp. 1–42.

Wilson, P. R., M. S. Johnstone, M. Neely, and D. Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review, *Proc. Int. Workshop on Memory Management.*

Zheng, C. and C. Thompson. 2000. PA-RISC to IA-64: Transparent Execution, No Recompilation, *IEEE Computer* (March), pp. 47–53.

Previous chapter
< Appendix A: Real Machines

Next chapter
Index >

INDEX

Abstraction, levels of, 1
Access flags, Java, 258
Addressing architecture, 77
ADORE system, 213
Amdahl systems, logical partitioning, 458, 465
Applets, 233
Application binary interface (ABI), 8, 83, 583–584
Application domains, 275
Application program interface (API), 8–9, 583
Java, 261–267
Application servers, 447
Architecture, 6–9
addressing, 77
register, 69–70
Aries system, 14, 118
Arithmetic, emulating, 74–75, 565
Array range checking, 318–320
Assists,
instruction emulation, 417–418
virtual machine monitor 418–419
Attributes
Java, 259
Microsoft common language infrastructure, 270–271
Backing store, 571
Backpointers, code cache, 135, 138
Basic block
dynamic versus static, 56–57
profile, 160
Big-endian byte order, 76
Binary classes, Java, 256–259
access flags, 258
attributes, 259
constant pool, 244–245, 258
field count and information, 259
interfaces, 258

methods count and information, 259
Super_Class, 258
This_Class, 258
Binary optimization, dynamic approach to, 180–181
backward and forward branches, 154
code optimizations, 201–208
code reordering, 186–201
common subexpression elimination, 147
compatibility, 181–184
compensation code, 150
control flow, 153–154, 159
data value predictability, 155
indirect jumps, 155
profiling, 148–150, 156–167
program behavior, 153–156
same-ISA, 15, 29, 208–218
staged emulation, 151–153
superblocks, 150, 173–178
techniques, 147–148
traces, 171–173
translation blocks, 167–180
tree groups, 178–180
Binary optimizers, 15
Binary rewriting, 515–519
Binary translation, 29, 49–52
incremental predecoding and, 55–62, 80, 82
locating program counter, 124–126
memory state, 127–128
precise guest state, determining, 124–128
register state, 126–127
Binary translators, dynamic, 13–15
Bootstrapping, 586–588
brk(), 129
Buffer maintenance, 48–49
Buffer overflow, 502–503
Bytecodes, 17
Java, 232, 246
MSIL, 271–273
Byte order, 76
Cache memory, 558–559

Callbacks, 130–132
Call graph, 309
Calling sequence, 309
Cellular Disco System, 475–476
fault containment, 480–481
memory borrowing, 482–483
memory mapping, 478–480
overview of, 477–478
recovery from faults, 483–485
Chaining, code cache, 64–66
Channels and channel command words, 414–415
Channel-to-channel adapter (CTCA), 421
Checkpoints, 194–195, 345–347
Child process, 584
CISC ISAs, 36, 38–49, 125, 329
Class loader subsystem, 284–286
clib, 9
Clustered computing, 590, 591–594
Code cache, 55, 85
chaining, 64–66
codesigned virtual machines and, 339–344
difference between conventional cache memory and, 134
FIFO, coarse-grained, 138–139
FIFO, fine-grained, 125, 138
flush when full, 136–137
implementations, 134
least recently used, 135–136
management, 63, 85, 133
performance, 139–140
preemptive flush, 137–138
replacement algorithms, 134–140
virtual machine monitor and, 393–395
Code discovery, 46, 52–54, 278
Code location problem, 54–55
Code optimizations
basic, 201–204
compatibility issues, 204–205
instruction-set-specific, 206–208
intersuperblock, 205–206
Code patching, 210, 211–213

Code reordering
branch instructions, 187, 188
condition code handling, 198–199
implementing a scheduling algorithm, 192–199
instructions, categories of, 187
join points, 187, 188, 190
memory update instructions, 187, 190
register update instructions, 187, 188–189
straight-line, 191–192
superblocks versus traces, 199–201
Codesigned virtual machines, 21–22
applications, 352–354
code caching, 339–344
early models, 330–331
IBM AS/400 system, 22, 330, 353, 357–367
IBM Daisy system, 334, 347, 353
input/output, 351–352
memory state mapping 333–337
precise traps, 344–351
role of, 332–333
self-referencing and -modifying code, 337–339
Transmeta Crusoe, 22, 24, 331, 338, 353, 354–357
Collective, Stanford, 526–532
Common language infrastructure (CLI), 17, 267–275
Common language runtime (CLR), 267
Common subexpression elimination, 147
Communication, inter-machine, 421–422
Compacting garbage collectors, 297–298
Compatibility
binary optimization and, 181–184
code optimization and, 204–205
extrinsic, 88, 184
intrinsic, 88, 184
process virtual machine, 87–95
state mapping and, 101–102
transparency, 88
Compensation code, 150, 196
Computation server, 549
Computer hardware, 554–561
Computing environments, migration of, 521–532

Concealed memory, 21–22, 334–337
Concurrent garbage collection, 300–302
Condition codes, 70–74, 198–199
Constant folding, 201
Constant pool, Java, 244–245, 258
Constant propagation, 201
Context block, 30
Continuation set, 175–176
Control flow, 153–154, 159
graph, 157, 160
Control Program (CP), 381–382, 395–396, 402, 414–415, 421
Control transfer
optimizations, 64–68
restricting, 516–517
Conversational monitor system (CMS), 370, 381
Copying garbage collectors, 299
Copy-on-write, 528–530
Copy propagation, 202–203
Counter decaying, 165
CPU timer, 384
CreateProcess, 141, 142
Critical instructions, 391, 392–393, 430–431
Daisy system, 334, 347, 353
Data formats, emulating, 74–75
Data server, 549
Data types, Java, 242–243
conversion, 249
Dead-assignment elimination, 202, 204
Decode-and-dispatch interpreter, 30, 80, 81
Device-driver interface, 408, 409–410
Devices, virtualizing
dedicated, 404–405
nonexistent physical devices, 407
partitioned, 405
shared, 405
spooled, 405–407
DIAGNOSE, 420
Digital Equipment Corp., FX!32
system, 14, 15, 46–49, 88, 118, 141, 142–145
Direct native execution, 382

Direct threaded interpretation, 37–38, 80, 81–82
Distributed shared memory (DSM), 590–591
Dual-address return address stack (DRAS), dual-address, 342–344
Dynamically linked library (DLL), 140, 142
Dynamic basic block, 56–57
Dynamic binary optimization. *See* Binary optimization, dynamic
Dynamic binary translators, 13–15
DynamoRIO system, 132, 139, 152, 209, 210, 214
See also RIO system
Dynamo system, 15, 112, 119, 129, 152, 209, 214–218
Edge profile, 161
Emulation, 12
basic, 305–306
binary translation and incremental predecoding, 55–62, 80, 82
decode-and-dispatch interpreter, 30, 80, 81
defined, 27
direct threaded interpretation, 37–38, 80, 81–82
engine, 85, 281, 283
exceptions, 86, 119–128
high-performance, 306–320
incremental predecoding and translation, 55–62
indirect threaded interpretation, 80, 81
manager (EM), 55, 58
memory architecture, 102–114
operating system, 128–133
same-ISA, 63–64
staged, 116–119, 151–153
Emulators, 13–15
Environments, migration of computing, 521–532
Epilog side table, 205–206
Escape analysis, 317
Event-driven modules, 511
Exception, use of term, 119
Exception emulation, 86
ABI invisible, 120
ABI visible, 119
exception detection, 120–121
interrupt handling, 121–122
precise guest state, determining, 122–128
Exceptions, precise, 277

Exceptions and errors, Java, 254–256
exec(), 141
Execution engine, 281, 283
Extrinsic compatibility, 88, 184
Fault(s)
containment, 480–481
recovery from, 483–485
Field count and information, Java, 259
FIFO, coarse-grained, 138–139
FIFO, fine-grained, 125, 138
File limits, enforcing, 133
Floating-point arithmetic, 74, 565
Flush-when-full, 136–137
Forth language, 29
Fujitsu, PrimePower systems, 456
Fully associative cache, 558–559
Fully qualified name, 284
Functional instructions, Java, 249–250
Fundamental Software Inc., FLEX-ES, 486
FX!32 system, 14, 15, 46–49, 88, 118, 141, 142–145
Garbage collection, 238–240, 283, 294–295
compacting, 297–298
copying, 299
generational, 300
incremental and concurrent, 300–302
mark-and-sweep, 296–297
optimizing, 320
root set, 302–303
summary of, 303
Generational garbage collectors, 300
getrlimit(), 133
Global memory, 243–244, 282
Globus project, 542–543
Grids
access issues, 540–541
characteristics of ideal, 539–542
compared with virtual machines, 543–546
computing model, evolution of, 537, 542–543
dependability, 539–540
evolution of, 535–538

implementing, on virtual machine systems, 546–551
resource sharing, 541–542, 544
use of, 546–551
Guard instructions, 311
Guests, 10, 28
improving performance, 419–422
integration of, into host system environment, 140–142
memory image, 85
precise guest state, determining, 122–128
state management, 375–377
virtualization with different host and, 485–496
Handle pool, 298
Handshaking, 420
Hardware, computer, 554–561
Hardware page table, 470
Hashed copy scheme, 531
Hazards, 597, 599
Heap-allocated objects, 316–318
Hewlett-Packard
-Compaq Superdome servers, 464, 465
Dynamo system, 15, 112, 119, 129, 152, 209, 214–218, 515
physical partitioning, 455–456
High-level language virtual machine (HLL VM), 15–17
High-level language virtual machine, architecture
common language interface, 267–275
ISA features, 275–279
Java, 241–261
object-oriented, 228–241
Pascal P-code, 225–228
platforms, 261–267
role of, 222–225
High-level language virtual machine, implementation
class loader, 284–286
components, 281–284
emulation, basic, 305–306
emulation, high-performance, 306–320
emulation/execution engine, 85, 281, 283
garbage collection, 238–240, 283, 294–303
memory, 281–282
native interface, 259–261, 283, 304–305

security, 286–294
Hitachi, logical partitioning, 458
Hoisting invariant checks, 319
Hoisting invariant expression out of a loop, 203–205
Host-based intrusion-detection systems (HIDS), 502–505
Hosted virtual machines, 379–381
input/output virtualization and, 412–414
VMware Virtual Platform, 426–435
Hosts, 10, 28
guest process integration into, 140–142
virtualization with different guest and, 485–496
Host-supported memory protection, 104–105
Hotspots, 159
Hybrid virtual machine system, 391
Hypervisors, 464, 474–475
services interface, 469–471
IBM
AS/400 system, 22, 330, 353, 357–367
Daisy system, 334, 347, 353
DK, 306
Interpretive Execution Facility (IEF), 424–426
iSeries servers, 464
Jikes Research Virtual Machine (RVM), 306, 320–327
PowerPC ISA, 35–36, 70, 74, 600–606
pSeries, 464, 600
System/38, 330, 357–358
System/360, 7–8, 381
System/370, 381–382, 395–396, 414–415, 427
System/390, logical partitioning, 458, 460–464, 473–474
zSeries, 465
IEEE floating point standard, 74
If conversion, 208
Image server, 549
Implementation
layers, 6–7
use of term, 6
Incremental garbage collection, 300–302
Incremental predecoding and translation, 55–62
Indirect jumps
binary optimization and, 155

map table lookups, 55–56
software prediction, 66–67, 340
Indirect threaded interpretation, 80, 81
Initialization
ISA system, 586–588
process virtual machine, 85
Inline caching, 66
Inlining
changing method calls, 310–312
guard instructions, 311
multiversioning and specialization, 312–313
partial procedure, 170
procedure, 169–170, 308–310
Input/output, codesigned virtual machines and, 351–352
IBM AS/400 system, 364–365
Input/output systems, 559–561
managing, 576–577, 582–583
system calls, 585
virtualization, 404–415, 431–435
Instruction checkpoints, 194–195
Instruction emulation, 114–119
assists, 417–418
Instruction set architecture (ISA), 7–8
examples, 600–611
resource management, 566–580
source, 13–14, 27–28
system, 8
target, 14, 27–28
user, 8, 561–566
virtual, 16–17
Instruction sets, 2
addressing architecture, 77
byte order, 76
condition codes, 70–74
conventional versus V-ISA, 278
data formats and arithmetic, 74–75, 565
IBM AS/400 system, 361–364
Java, 246–254
logically complete, 132
memory address resolution, 75

memory data alignment, 75–76
Pascal P-code, 227
register architectures, 69–70
Instrumentation-based profiling, 161–162
Intel IA-32, 427
arithmetic, 74
binary translation, 49–52
condition codes, 70, 71–74
description of, 606–611
EFLAGS, 71, 430, 431
high-performance interpretation, 46–49
interpretation of, 39–44
pop stack into flags register (POPF), 385–386, 391
threaded interpretation, 44–46
Intel IA-32 EL (execution layer), 14, 84, 119, 152
Intel IA-64 (IPF), 83, 186
Intel Itanium platforms, 83–84, 162
Intel VT-x (Vanderpool), 436–442
Interfaces
Java, 258
well-defined, 2–3
Internal Microprogrammed Interface (IMPI), 358
Internet Suspend/Resume (ISR), 524–526
Interpretation, 14, 29
basic, 29–32
of a complex instruction set, 38–49
decode-and-dispatch, 30, 80, 81
direct threaded, 37–38, 80, 81–82
indirect threaded, 80, 81
memory, 29–30
precise guest state, determining, 123
profiling during, 163–165
Interpretive Execution Facility (IEF), 424–426
Interpretive trap detection, 120
Interrupts
handling, 121–122, 577–580
hypervisor, 469
registers, 569
use of term, 119
Intersuperblock optimizations, 205–206