block can be restored via the hardware. At that point, interpretation is used to provide the precise exception state, just as with the software method. The basic sequence is illustrated in Figure 7.13b.

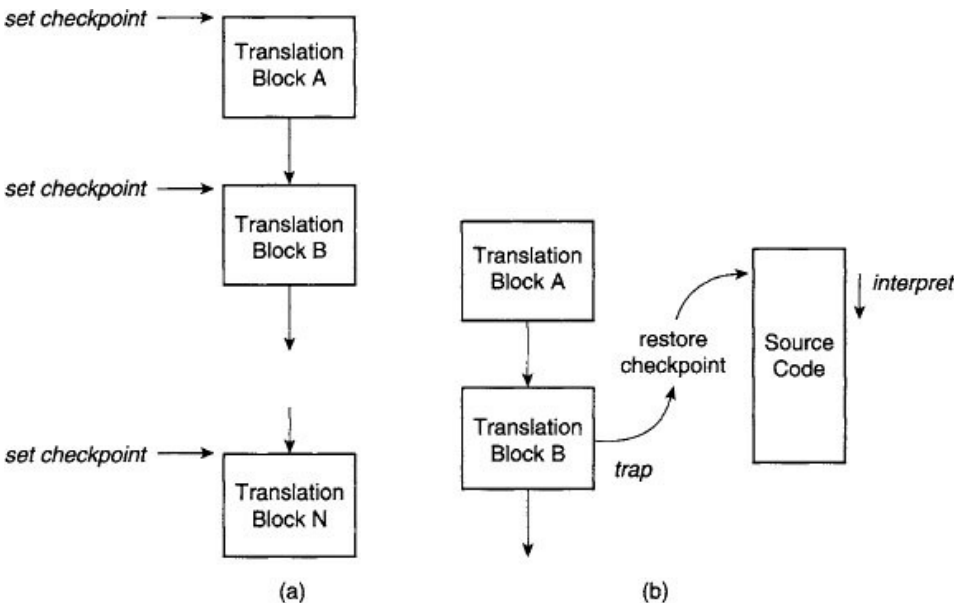


Figure 7.13 Hardware-Supported Checkpointing. *(a) Checkpoints are set at every translation block entry point. (b) When there is a trap, the checkpoint is restored, and interpretation begins at the beginning of the source code that formed the trapping translation block.*

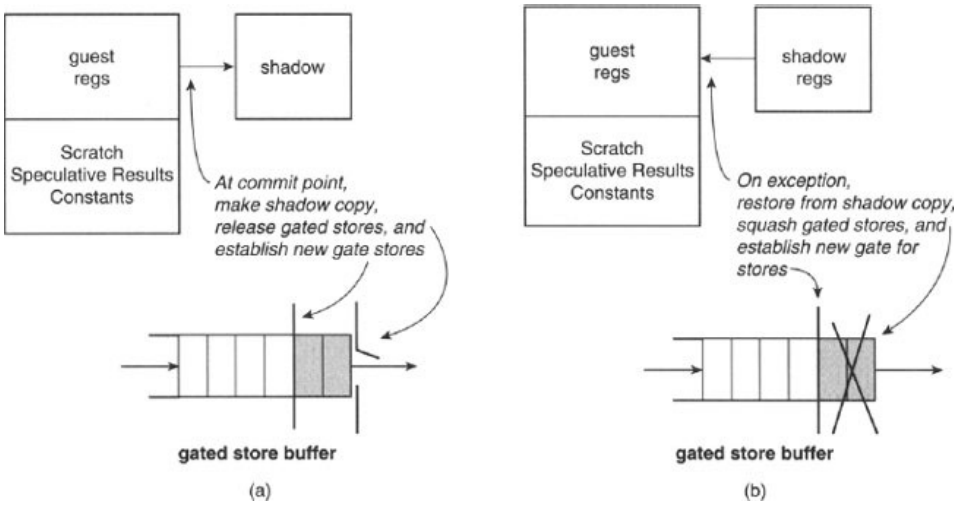


Figure 7.14 Checkpoint Hardware Operation. *(a) When a checkpoint is committed (and a new one is established), the gated store buffer is released and a new checkpoint is established. (b) If a trap is detected, the contents of the store buffer are flushed and the shadow copy is written back into the register file.*

The mechanism for supporting checkpoints is illustrated in Figure 7.14 (Klaiber 2000). When a new translation block is entered, the state from the previous block is committed, and a new checkpoint is

state from the previous block is committed, and a new checkpoint is set (**Figure 7.14**a). Setting register checkpoints is fairly easy. The host hardware can support a shadow copy of the guest registers (**Smith, Lam, and Horowitz 1990**), and registers are checkpointed by copying them *en masse* into the shadow registers. With proper hardware design (and circuit layout) this copying can be done in a very small number of cycles (as few as one). An alternative approach uses shadow registers at a finer granularity (**Nystrom et al. 2001**) and avoids discarding all the computation done in a code block when an exception is detected.

Checkpointing memory in a similar manner would be much harder because keeping a shadow copy of main memory would be very expensive, in both time and space. Hence, an alternative is to keep all memory store operations buffered in the processor until a translation block is exited (and its state changes can be committed) (**Klaiber 2000**). Refer to **Figure 7.14**a. At the time a checkpoint is committed and another is set, a gate is closed at the tail of the store buffer so that any instructions following the gate are inhibited from completing. Meanwhile, the gate at the head of the store buffer (established at the previous checkpoint) is opened so that stores can complete by writing to memory. These writes can occur in the background as computation proceeds. Note that all load instructions must check the store buffer as well as the data cache, in case a load is to an address that is in the buffer. This includes preceding, uncommitted stores in the same translation block and any residual, committed stores from a preceding block.

If an exception should occur during execution of the translation block (**Figure 7.14**b), then the buffered stores are flushed and the registers are restored from the shadow copy. At that point the state is the same as it was when the block was first entered. Then interpretation of the source code can take over to finish the job of locating the trapping instruction and providing a precise state.

An advantage of the hardware-supported checkpoint method is that the code inside a translation block can be reordered by software in any fashion (subject to true data dependences, of course), even when the block includes store instructions. Hence, the restrictions on code motion given in **Section 4.5.1** are relaxed. The only limitation is the fixed size of the store buffer, and this can constrain the translation block size in some cases (the number of stores in the block cannot exceed the store buffer size).

On the other hand, software reordering of loads and stores can lead to violation of memory ordering compatibility. In a shared-memory multiprocessor there are hazards associated with reordering loads and stores as observed by other processors (see Appendix Section A.7.3), which means that reordering may be limited in shared-memory multiprocessor implementations. The reason is that after the software reorders the instructions, the original order is lost; yet correct operation in parallel programs may depend on the original ordering. A method for dealing with multiprocessor memory ordering is described in a patent by members of the IBM Daisy project (**Altman et al. 2002**). This method is based on acquiring exclusive access to cache lines as stores enter the store buffer and then flushing the buffer and restarting if a different processor should attempt access to the line before the stores in the buffer are released.

7.4.2 Page Fault Compatibility

Thus far, we have not considered the recovery of the precise guest state when page faults occur. In most process VMs, page faults are not an issue because they are manifestations of OS policies, are handled by the OS, and are hidden from the process. A codesigned VM is a system VM, not a process VM, which means that the guest OS must observe exactly the same page faults as it would if it were running on a native platform. If the guest OS has mapped a logical page to a conventional real page, then it must not trigger a page fault; similarly, if the guest OS has not mapped a page and it is accessed, then

it must page fault. Implementing page fault compatibility is relatively simple for the data region of the guest's memory. If the host implements the same memory mapping as the guest (**Figure 7.6**) and lets the guest OS manage conventional memory, then page faults to the data region will be detected naturally by the host system. These faults will return control first to the VMM, which can use techniques as described in **Section 4.5.2** to determine if the fault is spurious (due to optimizations and/or code reordering) and to produce the correct, precise state before control is handed over to the guest OS.

Likewise, when interpreting instructions, the interpreter loads instructions from the conventional region of memory, so page faults for instruction fetches will occur naturally. For executing translated instructions from the code cache, however, the problem is more difficult because the guest instructions are not actually fetched; rather, translated instructions are fetched from concealed memory. The codesigned VM must therefore implement some mechanism that will trigger a page fault when a translated instruction is fetched from the code cache and the corresponding guest instruction *would have* caused a page fault had it been fetched on a native source platform. There are a number of possible approaches to solving this problem. One approach is *active,* another is *lazy.* Both approaches are discussed in the following paragraphs. Note that in this discussion we assume an architected page table and page faults. Special considerations for an architected TLB and TLB faults will be discussed in a separate subsection.

Active Page Fault Detection

The active approach to instruction page fault detection is to monitor potential page replacements by the guest operating system. When a source instruction page is replaced by the OS, all the translation blocks based on the replaced source page should be flushed from the code cache.

the code cache.

To determine when an instruction page is potentially being replaced by the guest OS, the VMM monitors the guest OS's modifications to the architected page table. This can be done by marking memory that holds the page table as "write-protected." Assuming the page table is architected, the region of memory that holds the page table can be identified by the VMM. Write-protecting pages is most easily done if VMM software manages the TLB. Then the write-protection information is added to the implementation of the TLB and does not require any changes to the architected page table in main memory. The VMM should also monitor any other operations that may change a virtual to real page mapping, e.g., changes to the page table pointer.

In addition, the VMM keeps a table of virtual page numbers for each page containing source instructions. At the time a block of source instructions is translated, an entry is made in the table (if it is not already in the table). Then whenever the page table mapping is modified, there is a trap (or jump) to the VMM, and the VMM determines if the page table entry for a source instruction page is being modified. If so, the VMM flushes all the translations in the code cache derived from that page. This requires a pair of side tables. One side table tracks all the translation blocks derived from each page; these are the translation blocks that must be flushed. The second table (or an extension to the first), keeps track of any link backpointers (see **Section 3.8.2**). The link pointers are changed so that they point into the VMM emulation manager. After this is done, an attempt to transfer control into the removed page will instead jump to the emulation manager. Subsequently, the emulation process will detect the instruction page fault when it attempts to emulate an instruction from the code page and it misses in the page table. Finally, if hardware structures containing SPC-to-TPC address translations are implemented, such as the JTLB and DRAS (**Section 7.3**), then these structures should be flushed of any entries corresponding to flushed

translations.

Lazy Page Fault Detection

With the lazy solution, when a source code page is replaced by the guest OS, the code cache is not immediately flushed of corresponding translations. Rather, it waits until there is an attempt to actually use the translated code. To accomplish this, every time the translated code crosses a source page boundary, the page table is probed to see if the mapping is the same as on the original translation (**Ebcioglu et al. 2001**).

The method is illustrated in **Figure 7.15**. First, it must be possible to determine when translated code crosses a source page boundary. This is most easily done if all the code in a translation block is derived from the same source page. If this is the case, then source pages boundaries are crossed only when there are link jumps, e.g., when block HIJ jumps to block KL in **Figure 7.15**. Otherwise, if translated code can come from different source pages, then the crossing points within the translated code should be distinct, at least with respect to precise state updates (the transition from E to F in **Figure 7.15**). In either case, at the time a source page boundary is crossed, the first instruction is a Verify_Translation instruction, inserted by the translator. This instruction probes the page table to see if the page mapping has changed. The Verify_Translation instruction provides the virtual address of the newly entered source page along with the real address at the time the block was translated. The instruction attempts to translate the virtual page number to a real page number and, if successful, compares it with the real address provided by the instruction. If the virtual page is mapped to real memory and the two real addresses match, then execution of the translated code proceeds. Otherwise, the VMM takes over and generates a page fault and/or translates the source code page that has just been entered.

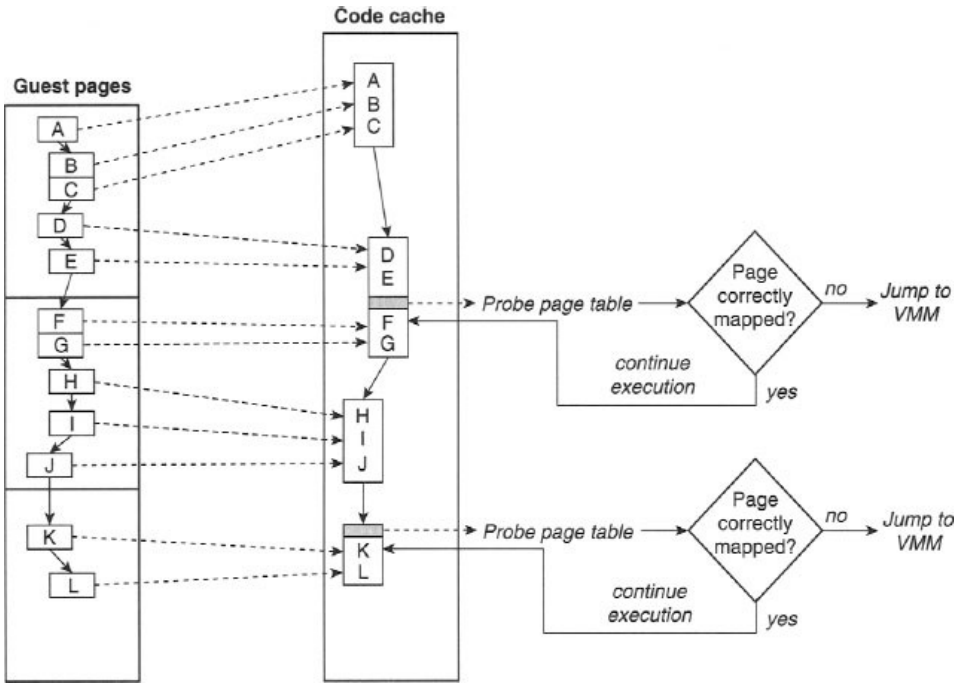


Figure 7.15 Detecting Instruction Page Faults via Instructions That Probe the Page Table.

Architected Translation Lookaside Buffers

With architected TLBs, both the active and lazy methods can be adapted in a straightforward way. But, because of performance issues, the active approach is somewhat problematic. If the active approach is used, then translations are flushed every time a TLB entry for a source code page is overwritten. Then the code must be retranslated if the TLB entry is later restored. If there is a fairly large number of source instruction pages and a lot of TLB activity for instruction pages, this approach could lead to a very high overhead. With the lazy approach, Verify_Translation instructions can be used, as before. The only difference is that they generate an architected TLB fault rather than a page fault when the translation is not present.

7.5 Input/Output

For the most part, implementing I/O in a codesigned VM is straight-

forward. If the VMM does not use any I/O devices itself, then all the guest device drivers can be run as is (although instructions in the driver code are emulated, just as all other guest instructions are). Any I/O instructions or memory-mapped I/O is simply "passed through" by the VMM so that the I/O system sees the same signals that it would if the guest were running on a native platform.

As noted in **Section 4.5.1**, memory-mapped I/O results in volatile memory locations, and the presence of volatile memory locations should inhibit optimizations involving removal and/or reordering of memory loads and stores to those locations. To identify code regions that access volatile memory, an access-protect bit, similar to the write-protect bit of **Section 7.2** can be added to the TLB. Then if a memory-mapped I/O routine performs loads or stores to an access-protected memory page, there is a trap to the VMM, which can then deoptimize translations containing the accesses to volatile memory in order to ensure that they all occur in the correct sequence (**Kelly, Cmelik, and Wing 1998**). As an additional enhancement, the implementation ISA can be extended to include special volatile versions of load and store instructions, which do not trap when the access-protect bit is set; if these are used in deoptimized I/O code, then their execution will proceed without trapping.

If a VMM uses no I/O devices, then it is restricted to reside entirely in concealed memory. This means, for example, that code translations cannot be cached to disk and reused as is done in the FX!32 system described in **Section 3.10**. All translations must be done from scratch each time a program is run; and when translated code is removed from the code cache, it is discarded and must be retranslated before it can be used again.

If the requirement of complete transparency can be relaxed somewhat, then concealed memory can be extended to secondary storage (disk). In order for this to be accomplished, there must be a portion of a disk (or possibly an entire disk) reserved for use by the VMM as

concealed secondary storage. The most straightforward way to do this is to provide a special disk driver that is VMM-aware (**Figure 7.16**). It is the loading of this special disk driver that breaks complete VMM transparency. This limits the use of the codesigned processor to those operating systems for which the special disk driver has been developed and deployed. The disk driver code can restrict the OS to that portion of the disk that is visible (actually, the guest OS is only informed of the nonconcealed region). Meanwhile, the VMM uses the disk driver to access its concealed region of disk.

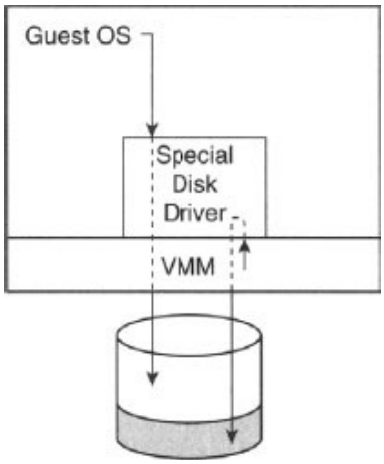


Figure 7.16 Concealed Secondary Storage. By adding a special disk driver, concealed secondary storage (shaded) can be implemented. Both the guest OS and VMM software can call the driver, but only the VMM can access the concealed region of the disk.

To avoid relaxing transparency even more, the VMM is entirely responsible for managing the concealed secondary memory. This implies that it is not backed up or archived, for example, so this restricts its use to caching code translations (or for paging VMM software). This large, persistent code cache would be useful for reducing initial start-up time for big programs (including the OS when it is booted). However, checks have to incorporated to make sure that any source code with cached translations has not changed since the time of the translation, possibly by keeping a copy of the original source code and performing an instruction-by-instruction check (**Conte, Sathaye, and Banerjia 1996**). Furthermore, the concealed secondary storage would only be useful for large blocks of translated codes—for small blocks it would probably be faster to trans-

lated codes—for small blocks it would probably be faster to translate from scratch than to make disk accesses. An interesting (and less transparent) disk-based code caching approach was proposed for dynamic rescheduling of VLIW binaries based on page-sized code blocks (**Conte and Sathaye 1995**).

7.6 Applying Codesigned Virtual Machines

Thus far, we have described a number of mechanisms that allow a codesigned VM to emulate full ISAs and execute instructions efficiently from a code cache. These mechanisms are only enablers; however, to be useful these mechanisms must be applied in codesigned VMs that provide real advantages, such as improved performance, power efficiency, and software flexibility. Advantages can be achieved at two levels, in effect at a macro level and at a micro level.

At the macro level, entirely new ISAs can be implemented. Some of the proposed and implemented target ISAs for codesigned VMs have been aimed at exposing instruction-level parallelism to hardware in a more efficient manner than with conventional instruction sets. Most conventional instruction sets still reflect the sequential model of execution prevalent decades ago. Consequently, hardware is burdened with discovering parallelism in the instruction stream at run time; this is the essence of modern superscalar processors.

Two of the better-known codesigned VMs, the Transmeta Crusoe (**Klaiber 2000**; **Halfhill 2000**) and the IBM Daisy/BOA (**Ebcioglu et al. 2001**; **Sathaye et al. 1999**), implement VLIW instruction sets as their target ISA. The codesigned software is responsible for finding independent instructions in the source binary and packing them together in the VLIWs. The advantage of a VLIW implementation is that it does not require the complexity (and power consumption) of an out-of-order issue unit as in most superscalar processors.

A proposal for a different style of codesigned VM has a similar goal—to simplify instruction issue logic—but it does this by exposing chains of dependent instructions to the hardware (rather than independent ones as with a VLIW) (**Kim and Smith 2003**). Then multiple in-order instruction issue units can be given the dependent chains of instructions. Each issue unit processes instructions in order, which simplifies hardware, but the multiple issue units can issue instructions out of order with respect to each other. A research project at the University of Illinois (**Merten et al. 2001**) provides a number of interesting codesigned hardware components targeted primarily at profiling and optimization, rather than ISA innovation.

The goal of the IBM AS/400 implementation (**Soltis 1996**) is to provide a high-level object-oriented source ISA. Besides being well suited for efficient support of object-oriented system and application software, this approach allows many of the hardware resource management mechanisms, e.g., page management, to be placed in the implementation-dependent VMM. Another advantage of the codesigned approach was demonstrated by the AS/400 designers when they successfully replaced the original host ISA (a proprietary CISC) with an extended PowerPC ISA and migrated users in a completely transparent way.

At the micro level, codesigned VMs permit the implementation of specific performance enhancements (which may be related to the selected host ISA). For example, in a VLIW platform, all code scheduling is performed by the software translation system, so the ability to reorder instructions is crucial to the success of a VLIW-based computer. Consequently, in the codesigned VMs that use VLIW target ISAs, there is special support to enable reordering of instructions, especially load and store instructions, because dependences among load and store instructions are difficult to determine statically (i.e., to *disambiguate* them).

Finally, an important feature of codesigned VMs is that implementation-dependent profiling hardware can be built in for use by dynamic translating/optimizing software (**Conte, Menezes, and Hirsch 1996**; **Heil and Smith 2000**; **Merten et al. 2000**). This profiling hardware can be matched to optimizations used by the codesigned software translation code as well as features in the microarchitecture.

To make the foregoing advantages more concrete, the next two sections give case studies of two of the more important codesigned VMs, the Transmeta Crusoe and the IBM AS/400.

7.7 Case Study: Transmeta Crusoe

Although some elements of the codesigned VM paradigm had been used previously, for example, to provide code portability, the Transmeta Crusoe broke new ground in using a codesigned VM for achieving power efficiency and design simplicity. The TM5000 series was introduced in early 2000; these processors are described here. The follow-on Efficeon (TM8000) series was introduced in 2003, it incorporates larger caches than the TM5000 series, and its VLIW instructions are twice as wide.

The Crusoe implements the Intel IA-32 ISA with an underlying proprietary VLIW instruction set that uses dynamic code translation and optimization, centered on a code cache. The microarchitecture of the TM5800 processor is illustrated in **Figure 7.17**. The VLIW (or a *molecule,* in Transmeta parlance) consists of four instructions (*atoms*). It is a classic VLIW in the sense that the individual instruction fields in the VLIW are dedicated to specific functions. These instruction fields feed a branch unit, a floating-point unit, an integer unit, and a load/store unit. For precise traps, the Crusoe uses the method described in **Section 7.4** and is based on shadow registers and a gated store buffer.

and a gated store buffer.

The memory hierarchy is illustrated in **Figure 7.18**. The VMM software is held in compressed form in a 512KB boot ROM. At the beginning of the boot process, this software is decompressed into a 2MB region of concealed memory. In the implementation shown, the code cache and various side tables consume 14MB. There are conventional 64KB level-1 instruction and data caches, but, in addition, there are two smaller memories dedicated to the VMM. These are for critical VMM code and data that the designer relies upon for low-latency access.

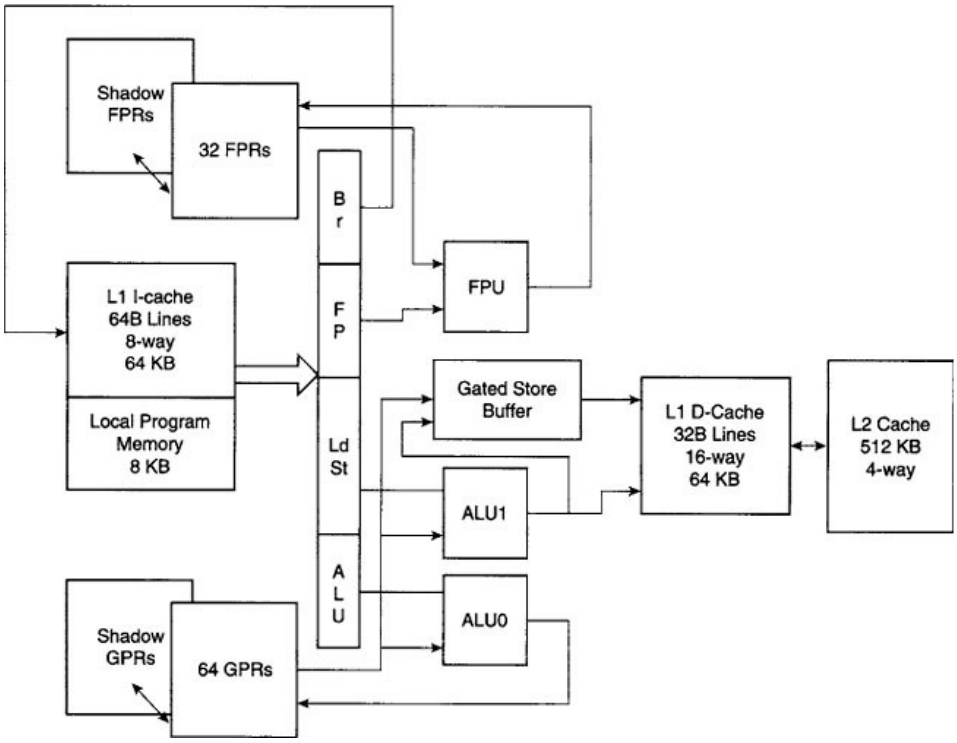


Figure 7.17 Transmeta Crusoe TM5800 Microarchitecture.

The big optimization challenges in the Crusoe involve breaking the complex IA-32 instructions into RISC-like micro-ops and then finding parallelism and scheduling them into VLIWs. Because an in-order VLIW microarchitecture is used, code reordering is a key part of the optimization process. To provide support for reordering, the Crusoe designers built some special features into the implementation ISA.

tion isn't.

Being able to move load instructions is perhaps the most important type of reordering, because load instructions take longer to execute than most other instructions, especially if there is a cache miss. That is, it is desirable to reorder a load instruction higher up in the instruction stream so that it executes earlier and can therefore eliminate or reduce the wait time of subsequent dependent instructions. For example, consider the code in **Figure 7.19**a, which contains two loads and two stores with an add instruction that consumes the two loaded values. It would be desirable to move the load instructions higher in the code sequence, as is shown in **Figure 7.19**b. Then there are other intervening instructions that can overlap with the execution time of the loads. Consequently, the time the add instruction might have to wait for its input operands is reduced, especially if one or both of the load instructions should miss in the data cache. However, moving the load instructions up may be unsafe, because the store

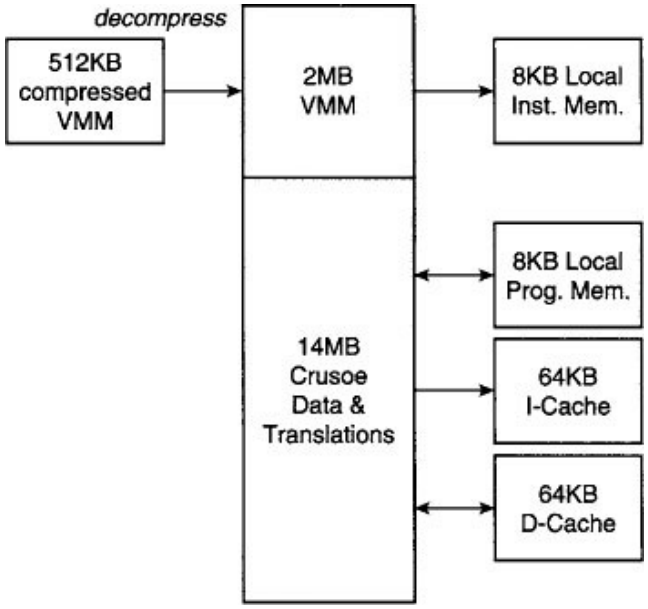


Figure 7.18 Transmeta Crusoe Memory Hierarchy.

Original code	*Rescheduled (unsafe)*	*Rescheduled (protected)*
`st 0(r1),r2`	`ld r3,0(r4)`	`ldp r3,0(r4) x`

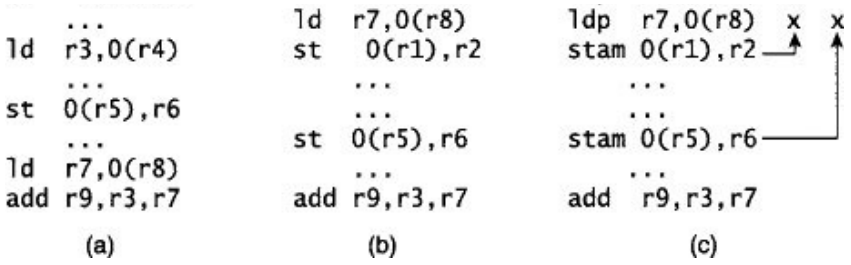


Figure 7.19 Safe Reordering of Memory Operations with Load-and-Protect and Store-under-Alias-Mask Instructions. *(a) Original code sequence; (b) unsafe reordering (due to unknown addresses); (c) reordering with protection from 1dp and stam instructions.*

instructions may be to the same memory address. For example, if the store to 0(rl) happens to be to the same address as the load from 0(r4), then in the reordered code the load will return the wrong data. Because the register values are often not known until run time and they may change over the course of the program's execution, the binary translator is not able to guarantee that the load and store addresses are never the same, so, to be safe, such a code reordering should be inhibited.

A codesign approach for detecting such memory conflicts is very well suited to this problem (**Gallagher et al. 1994**), and the Transmeta designers took such an approach by adding a pair of special instructions. The load-and-protect instruction (1dp) performs a load operation and records the address and size of the loaded data in an architected table. The store-under-alias-mask (stam) instruction performs a store and also contains a mask that identifies certain table entries—the ones set by preceding 1dp instructions. For each stam instruction, the masked 1dp instructions are illustrated in **Figure 7.19**c with arrows from the stam to the 1dp; the table itself is not shown. The second stam in the example identifies only the second load, because the first load was already ordered ahead of it. If a table entry indicated by the mask of a stam instruction detects an overlap between the 1dp's memory address and the stam's memory address, then there is a trap to the VMM. At that point, the VMM can handle the situation like it does most other traps: It can go back to the previous checkpoint and begin interpreting the original code in order to get the correct result. If a particular translation block re-

order to get the correct result. If a particular translation block repeatedly traps, then the VMM can reorder the code more conservatively, taking care to leave the overlapping load and store in their original sequence.

The 1dp/stam pair is an excellent example that illustrates the interplay between the codesigned hardware and software in a codesigned VM. In this case, where a VLIW is used, the ability to reorder code is very important, and the 1dp/stam instructions, in conjunction with the dynamic optimization code, facilitate code reordering.

7.8 Case Study: IBM AS/400

The architecture of the IBM AS/400 (derived from the earlier System/38) is perhaps not as widely known as other "textbook" computer architectures, but it is both highly innovative and commercially successful (**Soltis 1996**). The AS/400 includes a codesigned VM—in fact, the whole system, including system software, was designed from scratch using a fully integrated approach, so one could say the entire system was codesigned from top to bottom. Unlike the other codesigned VMs discussed in this chapter, however, the System/38 was not designed to support an existing, conventional ISA. Rather, the original System/38 designers invented a new, high-level ISA with the goals of software simplicity and machine (hardware) independence.

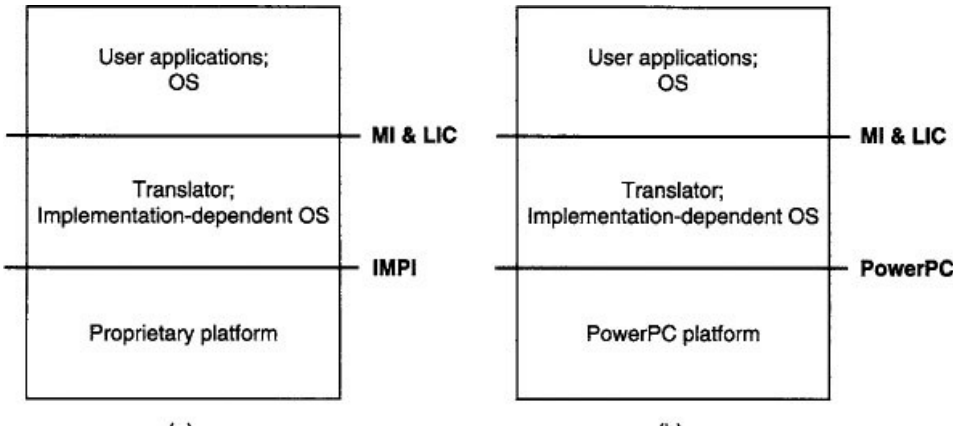




Figure 7.20 AS/400 Architectures. *(a) The original System/38 and early-generation AS/400 systems were based on a proprietary ISA (IMPI) and hardware platform. (b) More recent AS/400 systems are based on an extended PowerPC ISA and platform.*

The high-level instruction set is referred to as the *technology-independent machine interface,* or just MI for short. Supplementing the MI is a set of standard libraries, the Licensed Internal Code (LIC), which deals with the implementation-dependent aspects of resource management. The MI and LIC together from an ABI layer upon which all implementation-independent software runs.

In the original design, the underlying implementation architecture was based on a proprietary CISC ISA called the *internal microprogrammed interface* (IMPI)[1] (see **Figure 7.20**a). In developing a high-level ISA that is implementation independent, the objective was to provide for future changes in the underlying implementation, including the lower-level ISA. This feature has been exploited as the AS/400 evolved, first by extending the IMPI in a transparent way and then by a complete shift to an extended PowerPC ISA (**Figure 7.20**b).

Because the MI is at a higher level than a conventional ISA and was developed with the goal of hardware independence, the AS/400 architecture is similar in philosophy to the HLL VMs described in **Chapter 5** and follows the model illustrated in **Figure 5.1**. That is, the MI compiler is analogous to a conventional compiler frontend, and the MI translator is analogous to a conventional compiler backend. However, like other codesigned VMs, the AS/400 is a system VM. It supports not only language aspects of program execution but also a full system environment. The philosophy of dividing the implementation-independent and implementation-dependent software extends to the OS functions; for example, device drivers and memory management algorithms are in the implementation-dependent part of the OS. These functions are embodied in the licensed internal code, which uses the host ISA (IMPI or PowerPC).

The AS/400 uses an object-based ISA (the MI). So in this way it also follows a similar philosophy to the Java VM and the CLI (**Chapter 5**). However, in the AS/400, the MI itself is object based, in the sense that there are architected object types and type-specific instructions built into the MI. The programs that run above the MI do not necessarily have to be object based, however. In contrast, with Java and the CLI systems, the V-ISA includes architected primitives that are then capable of supporting the object-oriented programs that run above the V-ISA level; the only architected objects with their own instructions are arrays.

7.8.1 Memory Architecture

The MI has a memory architecture composed of objects. The objects are completely isolated from one another and can only be accessed via pointers; see **Figure 7.21**. There are a number of object types supported and used by MI instructions. The simplest type of object is simply called a *space,* which, as the name suggests, consists of a space for data. Other objects contain object-specific information, i.e., a "functional part" as well as space for constants and other object-related data. An object can be accessed only through a pointer created specifically for that object. To access a location inside a space, a space pointer or data pointer (for typed data) must be used. The functional part of an object is implementation dependent and is manipulated only by MI instructions specific to that type of object; the contents of the functional part are not otherwise visible above the MI. The MI-supported objects can be created, destroyed, and modified by programs.

Actual address values contained in pointers are not made visible to software above the MI level. Pointers can be used only for accessing data, and pointers cannot be modified with ordinary instructions. If an attempt is made to write to a memory location containing a pointer, then its ability to be used as a pointer is immediately de-

stroyed. The implementation of this feature is described shortly.

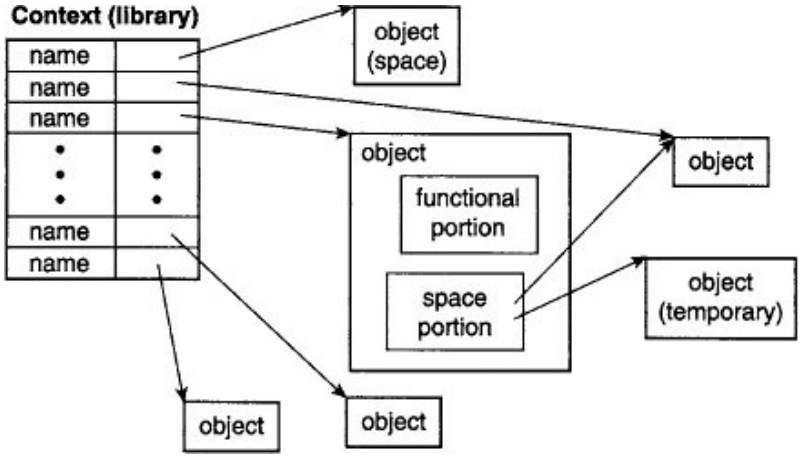


Figure 7.21 MI Memory Architecture. *Memory consists of objects, which may be either permanent (persistent) or temporary. A typical object contains a functional portion, accessible only via instructions specific to that type of object, and a space portion that can be modified with conventional instructions. The simplest object consists only of a space.*

Protecting the integrity of pointers is an essential part of any object-based system. It is interesting to contrast this method of pointer protection with the one used in the Java VM, where references are protected by carefully tracking their type so that illegal operations on references simply never occur in a properly verified program.

Although the MI object-based memory architecture is reminiscent of the heap used in Java and MSIL, a fundamental difference is that the AS/400 implements a system VM, so objects must persist over the lifetime of the system rather than just the lifetime of a process. There are no conventional files, from the perspective of the OS and applications. All storage, both in DRAM and disk, is one large object-based memory managed beneath the MI level. All long-term storage is done via permanent objects. Each of these permanent objects has a name that is contained in an MI object called a *context* (or a *library* in the operating system, OS/400). To get a pointer to an object, a program can specify a context and an object name; if the program is authorized to access the object, it is then given the correct pointer. This is similar to opening a file in a conventional system. Permanent objects persist in the system until they are explicitly destroyed.

In addition to permanent objects, a process can create temporary objects for its use as it runs. These temporary objects do not necessarily have a pointer held in a context, and they remain in the system until the system is rebooted. This ensures that there are no active programs that have a pointer to a temporary object at the time it is removed. There is no garbage collection in the AS/400, so this is the mechanism by which temporary space is returned to the system.

The implementation of the object-based memory, including all of memory management, is done entirely below the MI level. Main memory is based on the conventional PowerPC segment/page-mapping structure, as described in Appendix Section A.8.1. The object pointers are encoded in 128 bits, but the upper 64 bits contain type information and in some cases authorization information, and the lower 64 bits are essentially a 64-bit PowerPC virtual address.

The only significant extension to the PowerPC memory architecture is the adding of protection for object pointers, i.e., to prevent a program from overwriting a pointer in memory with an arbitrary value, and then later using it for an unauthorized memory access. This is accomplished by adding special instructions for loading and storing pointers and adding a “65th” bit to each memory doubleword. This 65th bit indicates whether the location contains a pointer (actually half of a 128-bit pointer). A load-pointer instruction checks this pointer bit; if set, it sets a condition flag that can be checked to verify that a valid pointer was loaded. A store-pointer instruction stores a pointer and sets the pointer bit(s). Every conventional (nonpointer) store always clears the pointer bit. Hence, any attempt to overwrite a pointer with a regular store results in what will be recognized as a nonpointer if it is ever accessed via a load-pointer instruction.

7.8.2 Instruction Set

The MI instructions are not intended to be executed directly, nor are they intended to be interpreted (although, in theory, they could be). They are essentially in a form that must be further compiled (translated) to the host platform's ISA before they can be directly executed. In the original System/38 this was the IMPI; in the more recent systems, it is an extended PowerPC ISA. The MI ISA performs fairly conventional operations that can operate on normal data types held in a space. It also has instructions that operate on MI-defined objects.

The MI instruction format, shown in **Figure 7.22**a, consists of an opcode and operand fields. The opcode is two bytes; the opcode extender (described shortly) is also two bytes. There are zero to *n* operand fields, each three bytes long (in the original System/38, two bytes). For basic arithmetic and logical instructions there is a long format and a short format. The long format is *operand1 ← operand2 op operand3,* where "op" is the operation being performed. The short form is *operand1 ← operand1 op operand2,* i.e., the first operand is both a source and a destination.

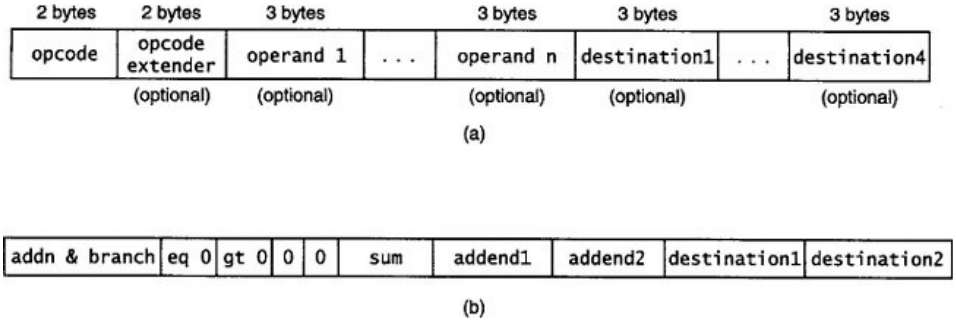


Figure 7.22 AS/400 Instructions. *(a) The general instruction format; (b) an example instruction that performs an addition and a multiway conditional branch.*

An interesting feature of the MI is that every instruction can include a four-way conditional branch or can evaluate up to four predicate values (called *indicators* in AS/400). These are implemented via the extended opcodes. If there is an opcode extender, then the instruction also has up to four branch destinations (or addresses for indicator values). The 16-bit extended opcode is divided into four 4-bit

fields. Each of the 4-bit fields encodes a condition, such as *greater than zero, equal to zero, less than or equal to zero,* etc. If the opcode extension is for a branch and the result of the instruction matches the first test condition, then there is a branch to the first branch destination; if it matches the second condition, it branches to the second destination, etc. A condition of zero indicates no test (and no destination is given). Similarly, for an indicator-setting instruction, the four destination fields correspond to up to four locations in memory where true/false predicate flags are stored.

For example, the instruction in **Figure 7.22**b does the following—it performs an addition on *addend1* and *addend2,* and the result is placed in *sum.* Then if the sum is zero, it branches to *destination1;* if the sum is greater than zero, it branches to *destination2;* otherwise, there is no branch, and the next instruction is executed. This instruction takes a total of 19 bytes, but it is important to keep in mind that this is the architected *representation* for the instruction. Before it is actually executed, it is translated to an implementation-dependent form that will be more compact, and it does the work of multiple RISC instructions. This instruction, for example, would likely compile to three PowerPC instructions, an add followed by two conditional branches.

The next matter is the way the operands are specified. The operand fields point into a two-level table that contains descriptors of the operands; this is called the *object definition table,* or ODT. As used here, the term *object* is being overloaded; these so-called objects are different from the true architected objects to be discussed later. The ODT, along with an instruction referencing it, is illustrated in **Figure 7.23**. The first-level table is the ODT direction vector (or ODV). This table consists of up to 16 million fixed-length fields that are directly addressed by the 3-byte instruction operand fields. The entries in the ODV contain a description of an operand type along with an optional pointer to the second-level table, the ODT entry string (OES).

The OES is used for those operand descriptors that don't fit in the fixed-length ODV entry.

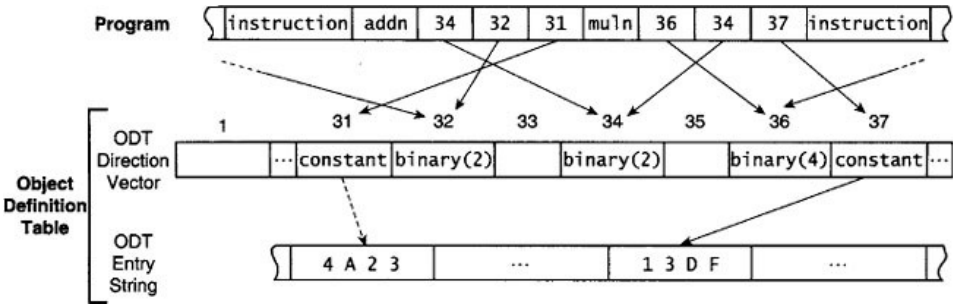


Figure 7.23 Specifying AS/400 Operands. *AS/400 operands are specified by an object definition table (ODT).*

In the example shown in **Figure 7.23**, two instructions in a program are shown along the top. The first instruction is an addn (add numeric). Its first operand is specified in ODT entry 32 and the other is in ODT entry 31. These operands are 2-byte binary (two's complement) numbers. The second operand is a constant (x4A23), and the pointer in ODT entry 31 points to the OES entry that holds the constant value. The result is also a 2-byte number (as indicated by entry 34). The second instruction takes the result of the first as one operand and a second constant (x13DF) as the second and multiplies them to form a 4-byte binary result. In these arithmetic instructions the opcodes, add numeric and multiply numeric, are generic; the actual type of operation (two's complement, decimal, or floating point) is determined from the types of the operands given in the ODT.

Note that the entries in the ODT indicate the types of operands and the data flow (e.g., that the result of the addn is an input to the mu1n). The actual storage locations for operands are assigned only after the MI is translated to the implementation instructions. At that time the specified operands may be assigned to registers, to a VMM-managed region of memory, or both. The constant operands would likely be translated into immediate fields of implementation instructions. This is very much like the register assignment process ordinarily performed in a compiler backend; here, it is deferred to the

VMM translator.

As pointed out earlier, all logical memory resides within objects; therefore accessing operands in memory must be done through pointers. Pointers are another operand type that can appear in the ODT. A pointer is returned when an object is created or when a symbolic pointer (e.g., as would be held in a context) is resolved.

As noted earlier, some instructions are object specific. The list of MI-supported objects is given below. Note that because the AS/400 was targeted at commercial applications, it has a number of objects intended to support databases. The operating system can use these objects directly for its purposes (the OS is also object based), or it can use these objects as primitives to build OS-defined objects.

Access group, Context—objects used for object management

Authorization list, User profile—objects used for supporting security

Byte string space, Data space, Dump space, Space—objects whose primary use is to hold data

Commit block, Cursor, Data space index, Dictionary, Index, Journal port, Journal space—objects whose primary use is to support database operations

Class-of-service description, Mode descriptor, Process control space, Queue, User profile—objects for process management, used by the OS

Controller description, Logical unit descriptor, Network description—objects used to support I/O

Module, Program—objects that contain translated code

7.8.3 Input/Output

Because of the focus on commercial applications, the I/O system in the AS/400 is a very elaborate one, based on I/O processors (IOPs), separate from the central processor. Consistent with the overall AS/400 philosophy, I/O is divided into implementation-independent and implementation-dependent parts, with the MI separating the two. The presence of IOPs simplifies the task of pushing the device-dependent aspects out of the central processor (and well below the MI layer).

At the level of the MI there is no secondary (disk) storage; rather, it is part of the unified memory architecture. In other words, disks are not considered to be part of the I/O system, and they have no presence above the MI level. All disk management software, drivers, etc. exist in the implementation-dependent part of the system. It should be pointed out that taking this approach is simplified in an environment where a single company is designing the entire system, i.e., where there is a high degree of vertical integration. The I/O devices visible above the MI layer are divided into generic types, for example, printers, keyboards, graphics displays. The logical characteristics of each of these general types are characterized by MI-level system objects called *logical unit descriptors.* Similarly, device controllers that may be present in the system can each control multiple devices. Two other MI-level objects, the controller descriptor object and network descriptor object, contain the logical characteristics of device controllers and network interfaces. The operating system interacts with software below the MI level (and, in effect, with the I/O devices) through instructions that operate on the MI-level objects.

7.8.4 The Processor Resource

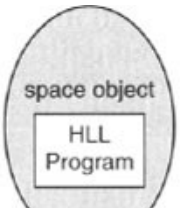
Above the MI level, the operating system establishes overall usage policies, for example, what the priorities the processes should be and what quotas of various resources should be. The actual imple-

and what quotas of various resources should be. The actual implementation of the policy is below the MI layer. That is, the implementation-dependent part of the system actually manages the scheduling queues.

Overall, many conventional operating system functions are done below the MI. This again reflects the overall integrated structure of the AS/400 design environment. When both the platform designers and the OS designers work at the same company (and largely at the same company site), it is easier to divide the OS functions in this manner.

7.8.5 Code Translation and Concealment

One of the more interesting aspects of the AS/400 is the way that it manages programs while concealing the implementation-dependent details. The process of compiling and translating a program is illustrated in **Figure 7.24**. First, a high-level language program is compiled, with the resulting MI code and object descriptor table being placed into a space object. Here, it is referred to as a *template* for an object yet to be created. Then a create program instruction is executed. One of its source operands is a pointer to the space object holding the program template. This instruction is "executed," which in fact results in the complete translation of the template contents into a program object containing the implementation-dependent executable file, e.g., PowerPC code. The create program instruction returns a result operand, which is a pointer to the program object containing the translated code. After being created, a program can be executed, and it can be maintained persistently or temporarily, just as any other object.



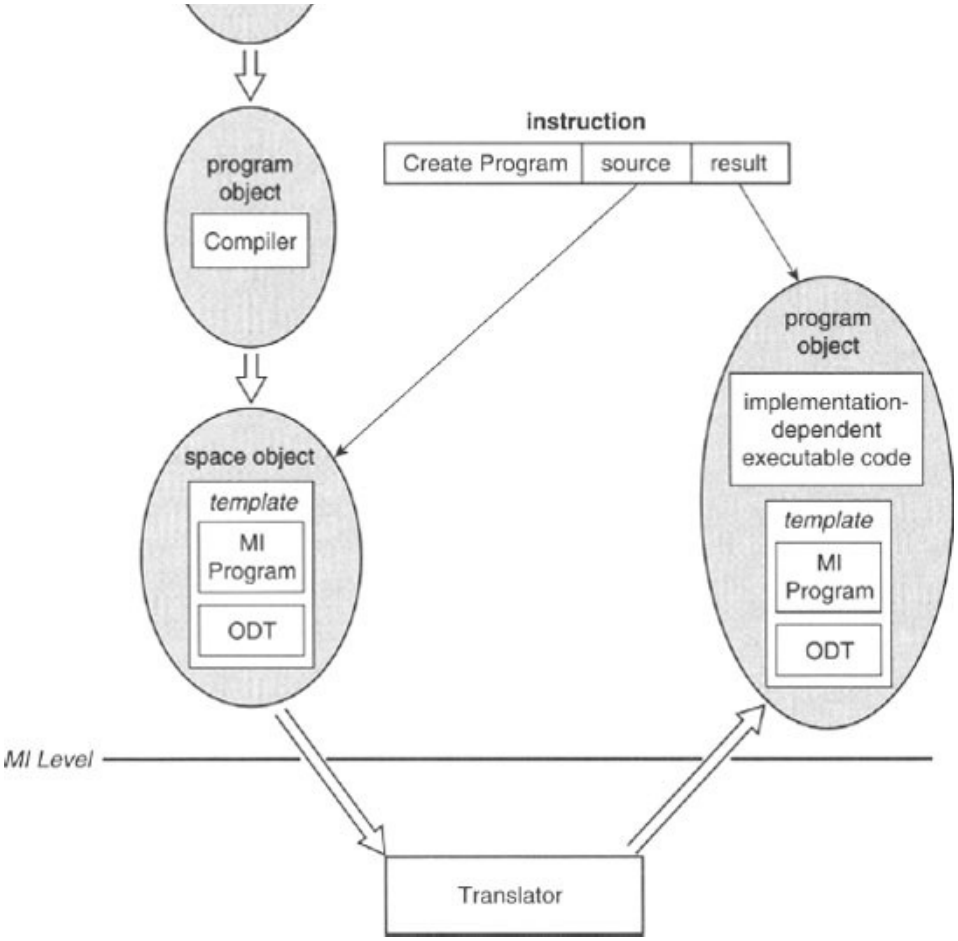


Figure 7.24 Compiling and Translating a Program on the AS/400. *The steps of compiling an HLL program and translating it into an implementation-dependent form.*

The contents of the program object cannot be directly observed above the MI level; i.e., the contents are concealed, just as with any other object. However, there are times when it may be desirable to look at the program code, for example, when debugging. To accomplish this, the original program template is also kept as part of the program object. Then the original program can be *materialized,* that is, given back to the user in its original, machine-independent form.

This feature yields another interesting capability. Say that a program has been compiled on a given hardware platform and is stored as a persistent object. Then the hardware platform is changed, as when IBM changed the AS/400 platform to PowerPC. Because the original template is stored in the program object, the platform switch is transparent to the user; in particular, the user

does not have to recreate the program object for the new host platform. Rather, when the user attempts to execute the program object on the new platform, the software below the MI layer will recognize that the translated code is for a different platform and use the original program template to produce a translation for the new host platform and place it in the program object.

Finally, it is interesting to compare this overall approach to implementation-dependent concealment with that used by the other codesigned VMs discussed in this chapter. The other codesigned VMs perform both translation and caching of translated code in a concealed manner. This means that each time a program is executed, it must be translated from scratch and then held in a code cache until the program terminates, at which time the code cache contents are discarded. In the AS/400, the fact that programs must be translated is not concealed, yet the actual translated code is concealed (within an object). Because the translation process is not concealed, the system can maintain translated programs in a persistent way, by treating them like any other persistent objects.

7.9 Summary

To some extent, modern codesigned VMs, as exemplified by the Transmeta Crusoe, should still be considered a somewhat experimental approach to computer design. This approach is based heavily on dynamic translation to a concealed code cache. Translations do not persist between successive program runs, so translation overhead is a major concern. On the other hand, it takes tens of milliseconds to access the disk when loading a conventional program or when performing other loading and linking tasks, possibly including additional dynamic loading and linking, as the program runs. When viewed from this perspective, the time required for dynamic translation does not appear so great, and, in a sense, it could be considered part of the loading process. Furthermore, the shift to

be considered part of the loading process. Furthermore, the shift to just-in-time compilation for HLL VMs will tend to make this approach appear more conventional as time passes.

The earlier AS/400 object-based approach, although successful and still used in the IBM iSeries machines, has not been copied in other commercial products. It is based on an unconventional separation of hardware and software functionality that probably works best in a completely vertically integrated system development environment. This design environment is much less common today than it was when the AS/400 (System/38) was first developed. A recent research project at the University of Illinois (**Adve et al. 2003**) advocates a new V-ISA that is in some respects similar to the AS/400 MI; it uses object-based memory architecture and a register-intensive instruction set. A primary goal is passing performance-critical information down to the hardware, where it can be more readily used. In other related work, the DELI project (Desoli 2002), builds on the earlier HP Dynamo project, and provides a callable interface into the emulation layer, thereby allowing application or system software the ability to interact with the emulation process.

Thus far the only real example of a modern codesigned VM is the Transmeta Crusoe, although the IBM Daisy/BOA project was carried far into software development and simulation. Both of these systems use a target ISA based on VLIW, and the suitability of VLIW for general-purpose computing remains a topic for debate. The Transmeta processors are targeted at power efficiency; their performance is not at the same level as the high-end superscalar processors that dynamically reorder instructions. Because most contemporary processors use out-of-order superscalar designs, when evaluating performance it is difficult to separate the codesigned aspect from the VLIW aspect to reach any conclusions regarding the overall suitability of the codesigned VM approach. The bottom line is that the codesigned VM approach remains a very interesting technology whose full potential remains to be fully determined.

Finally, an interesting future application of codesigned VMs is to target multiple source ISAs with the same platform. That is, to develop a single processor capable of executing software from multiple, different ISAs, depending on the emulation software that has been loaded into concealed memory. These *convergence architectures* (**Gschwind et al. 2000**) would, among other things, allow computers in server farms to be dynamically customized to fit the software they are called upon to execute.

---

1. Calling the codesigned software *micro-code* indicates that it was considered to be part of the hardware design, as opposed to being considered conventional software.

Previous chapter
< Chapter 6: High-Level Language Virtual Machine Implementation

Next chapter
Chapter 8: System Virtual Machines >

Chapter Eight

System Virtual Machines

It was observed very early that many of the hardware resources in a typical computer system are underutilized. The concept of *time sharing* was developed to improve resource utilization by allowing multiple users to access a single computer system simultaneously, with each user being given the illusion of having access to a full set of system resources. In order to create this illusion, a multiprogramming operating system essentially implements a process virtual machine for each application-level program and switches resources among the programs on a time-shared basis.

System virtual machines take this concept one step further by providing a similar illusion for complete systems. A system VM environment is capable of supporting multiple system images simultaneously, each running its own operating system and associated application programs. Each operating system controls and manages a set of virtualized hardware resources, as illustrated in **Figure 8.1**. The virtual resources include a processor (or processors), storage resources, and peripheral devices needed to perform input and output (I/O) for the system.

In a system VM environment, real resources of the *host* platform are shared among the *guest* system VMs, with a layer of software, the virtual machine monitor (VMM), managing the allocation of, and access to, the hardware resources of the host platform. The VMM owns the real system resources and makes them available to one or more guest operating systems that alternately execute on the same hardware. Thus, a guest operating system in a virtual machine environment is given the illusion of owning the resources it subsequently allocates to its various user programs

allocates to its various user programs.

Each virtual resource may or may not have a corresponding physical resource. When a corresponding physical resource is available, the VMM decides how access will be provided to virtual machines requiring its use; e.g., the resource may be partitioned or may be time shared. When a virtual resource does not have a matching physical resource, the VMM may emulate the action of the desired resource, typically via a combination of software and other resources that are physically available on the host platform.

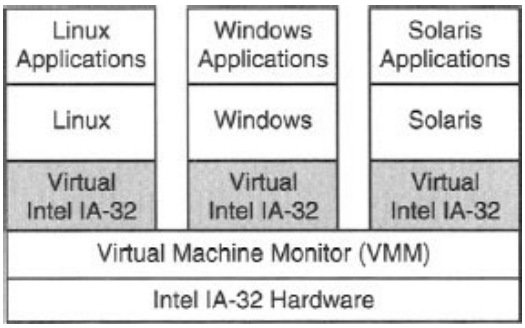


Figure 8.1 Example System VM Environment. *In this example, the Intel IA-32 platform is capable of simultaneously running the Linux, Windows, and Solaris operating systems, along with their applications.*

System virtual machines have a number of useful applications, some historical, some relevant today, and many that are likely to be important in the future. Some of these applications are described next.

- **Implementing multiprogramming:** Using multiple single-user virtual machines is a simple way of providing multiprogramming and time sharing without requiring a complete multiprogramming OS. Each user in a multiuser environment is given the illusion of having an entire machine, including the CPU, memory, and peripherals. Many of the users of VMs on early mainframe computers, such as the IBM System/370 and its successors, used virtualization as an alternative to a time-sharing system. These users ran a simple, compact, and efficient single-user operating system called the *conversational monitor system* (CMS), instead of the monolithic time-sharing operating systems available at the time.
- **Multiple single-application virtual machines:** This is an extension of the multiple single-user virtual machine concept. Running each application in its

multiple single-user virtual machine concept. Running each application in its own virtual machine increases the robustness of the system. Erratic behavior of one application and its associated OS on a conventional system could bring down the machine. If this happens on a virtual machine, it is less likely to affect the operation of another application running on a different virtual machine. This may be useful in situations where a new application might corrupt the entire system because of either bugs or viruses.

- **Multiple secure environments:** A system VM provides a *sandbox* that isolates one system environment from other environments, thus ensuring a level of security that may not be provided by a single operating system. For example, a user accustomed to running applications on a private machine may be reluctant to transfer the applications to a Web server unless there is some guarantee that the user's resources and activity cannot be accessed or monitored by other users of the same server. In such a situation, a virtual machine can be used to provide an environment that, for all practical purposes, is isolated from the others and through which it is not possible for one user to observe or change another's data and activity.
- **Managed application environments:** In an organization that provides a core set of supported applications to users who also want to install their own applications, VM technology allows the core set of applications to be placed on one virtual machine, protected from other virtual machines on which users install new applications or develop their own programs.
- **Mixed-OS environments:** A single hardware platform can support two different operating systems concurrently. A user may prefer office productivity tools that run on one operating system, for example, but prefer another operating system for application development. The user can do both by installing two virtual machines on a single hardware platform, one for each operating system.
- **Legacy applications:** When a new version of an OS is released, it is not uncommon for OS developers to support seldom-used features with degraded performance while supporting new features that allow most new applications to run with improved performance. In this situation, users who need to run old applications without degraded performance can use one virtual machine to run the "legacy" application under the old OS while using another VM to take advantage of the presumably improved features in the newer version of the OS.
- **Multiplatform application development:** Software developers often need to implement and support software that works on multiple operating systems. Virtual machine technology allows software developers to test their software using multiple virtual machines on the same hardware platform. This is both more convenient and more cost effective than having multiple sets of hardware, one for each OS.
- **New system transition:** A system virtual machine allows a user to migrate to a

- **New system transition:** A system virtual machine allows a user to migrate to a new operating system in a gradual way. The user can try out a new operating system on one virtual machine even while the old operating system is running applications on a different virtual machine. The old operating system can be removed from the system after it has been ascertained that all relevant applications work correctly on the new operating system.
- **System software development:** In large systems, it is sometimes necessary to develop (or test) new system software while the same system is being used for running important applications. In this situation it could be very costly if a bug in the code under development were to bring down the entire system. This situation can be avoided by encapsulating the development environment on a virtual machine separate from the virtual machine where the production-level applications are running.
- **Operating system training:** It is often necessary to demonstrate techniques and effects of changing parameters or policies when training system management personnel. Such training is better done on a copy of the operating system running on a virtual machine than on the operating system running on the real machine so that other users of the operating system do not suffer from unexpected effects.
- **Help desk support:** Help desk service personnel can bring up a virtual machine that emulates the hardware configuration of a client in order to determine the nature and cause of problems the client may be encountering. It is not necessary to have all possible configurations of hardware physically available—virtual machine technology allows a single hardware platform to be set to any of several different configurations.
- **Operating system instrumentation:** Running the operating system on a virtual machine allows the virtual machine monitor to instrument accesses to hardware resources selectively. Not only can all events of a particular type, e.g., page faults, be counted, they can also be logged with detailed information about their nature, their origin, and how the requests were satisfied. Moreover, the programming for all such instrumentation and measurements can be encapsulated outside the operating system, communicating solely with the virtual machine monitor. Today, it is quite common for operating system researchers to conduct most of their experiments using virtual machines rather than directly on the hardware platform. **Keefe (1968)** describes early work on IBM mainframes on system evaluation using a virtual machine. The User Mode Linux system (UMLinux 2003) also uses virtual machine technology to test the fault tolerance of a Linux system.
- **Event monitoring:** Some virtual machines provide capabilities that cannot be performed on a native system. For example, virtual machines lend themselves better to providing traces of execution or dumps of the machine state at points of interest. Similarly, the ability of virtual machines to replay a system execution

interest. Similarly, the ability of virtual machines to replay a system execution from some saved state is useful for analyzing the behavior of a system to understand unexpected behavior.

- **System encapsulation:** System virtual machines provide a convenient way of encapsulating the state of the entire machine. This is useful for checkpointing machine state, with a view to resuming execution at a different time or even on a different host machine.

We are nearly ready to describe system VMs in more detail. Before doing so, however, we first point out that this chapter considers a very important class of system VMs but not all system VMs. In particular, as described in the taxonomy given in **Section 1.5**, this chapter will focus on system VMs where the ISA of the host and guest are the same. Furthermore, most of the attention will be given to uniprocessor systems. Important aspects of VM implementation related to other types of system VMs can be found elsewhere in this book, as summarized shortly.

Multiprocessor virtualization is a simple extension of uniprocessor virtualization. However, there are some special considerations when a multiprocessor is being virtualized. One consideration is that with a single processor, virtualization must be done by time sharing the single processor resource. With multiprocessors, however, virtualization can be achieved by partitioning the multiple hardware processors among the virtual systems. The choice between time-sharing processors and partitioning (or using both) opens up a number of additional design alternatives when multiprocessors are being virtualized.

Another important consideration for shared-memory multiprocessors, particularly when the host and guest ISAs are different, is the memory models supported by the guest and host platforms. The two aspects of memory models that are crucial are memory coherence and memory consistency. Both of these involve the way in which memory operations performed by one processor are observed by other processors in the systems. If the host system does not support the same coherence and/or consistency models as the guest as-

sumes, then the virtualization process may need to take special steps to ensure compatibility. We discuss considerations that are specific to multiprocessor virtualization in **Chapter 9**.

When the guest and host ISAs are different, software in the VMM can emulate the virtual ISA using the techniques described in **Chapters 2**-**4**. **Chapter 9** will address some of the system VM issues that arise where the ISA of the guest is different from that of the host.

8.1 Key Concepts

In this section we provide an introduction to the various parts of a hardware platform, with a view to understanding how virtualization can be achieved.

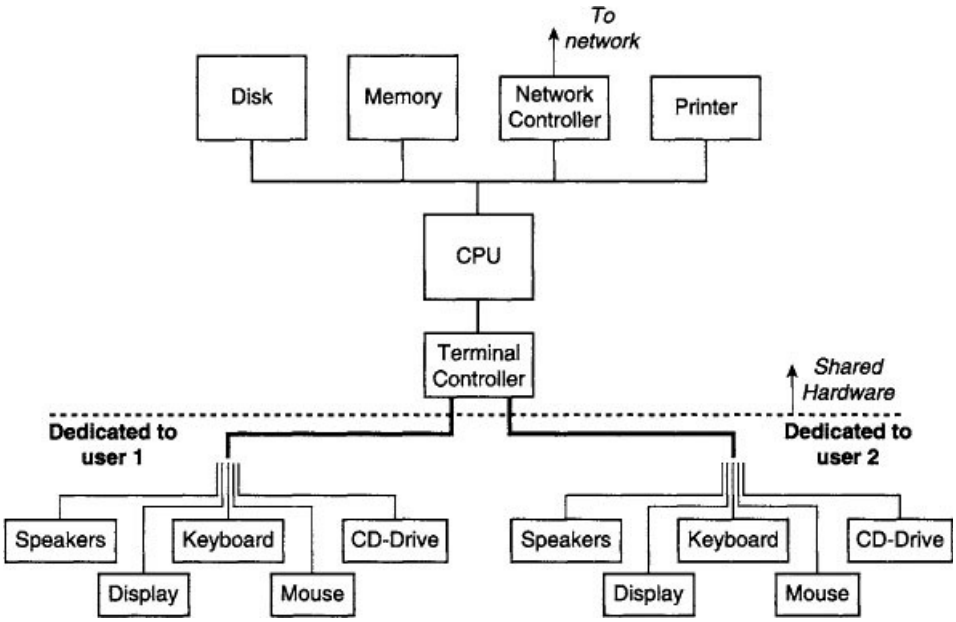


Figure 8.2 Hardware Replication for a Two-User Virtual Machine System. *The terminal controller collects and tags the requests from multiple users. If such a piece of hardware is not available, the computer backplane must have enough slots to accommodate the adapters for both sets of personal devices.*

8.1.1 Outward Appearance

The illusion of multiple machines is an important part of many VM systems. This illusion can be accomplished purely by software means or, in some cases, through a replication of a subset of the hardware resources. For example, there may be cases where the physical resources used directly by individual users—namely, the keyboard, display, and personal peripheral devices, such as a CD ROM drive—are replicated, with one dedicated set for each user, while the rest of the hardware is shared via virtualization software. This situation is shown in **Figure 8.2**.

Alternatively, when a single user wishes to run two operating systems on the same hardware, there is no need to replicate any of the physical devices. Yet the user can be provided with the illusion of running both operating systems simultaneously. By using a hardware switch or, more typically, by entering a special key sequence on the keyboard, all the personal devices can be switched from one virtual machine to the other.

In some system VM environments, one of the operating systems may be considered more important than the other. The user interface of the first provides a window displaying the full user interface of the second. Interactions with applications running on the second operating system take place within this window. In a hosted virtual machine, for example, if the host operating system were Windows, a graphical window could be established on the desktop interface to interact with a virtual machine on the same platform. This solution is similar to the DOS window that may be brought up within the Windows desktop interface to run legacy DOS applications.

8.1.2 State Management

The architected state of a computer is contained in and maintained by the hardware resources of the machine. From a performance point of view, all state-holding hardware resources are not equivalent. There is usually an architected hierarchy of state resources,

ranging from registers at the top of the hierarchy to secondary storage (e.g., disks) at the lower end of the hierarchy. In a VM system, each virtual machine has its own architected state information, and there may or may not be adequate resources in a host hardware platform to map each element of a guest's state to its natural level in the host's memory hierarchy. For example, a guest's register state may actually be held in the main memory of the host platform as part of a register context block.

In normal operation, the VMM periodically switches control among the guest VMs. Regardless of where the state is actually held, as operations on the guest's state are performed (e.g., via instruction execution), the state maintained on the host machine should be modified just as it would be on the guest's native platform. This is essentially the VM isomorphism discussed in the book's introduction and illustrated in **Figure 1.2**. There are two basic ways of managing guest state so that this is accomplished.

One way is to employ a level of indirection, by holding the state for each guest in fixed locations in the host's memory hierarchy with a VMM-managed pointer indicating the guest state that is currently active. As the VMM switches among guest systems, it changes the pointer to match the current guest. This is illustrated in **Figure 8.3**a, where the pointer actually points to the register context block of the currently active guest virtual machine. This is analogous to the situation in virtual memory systems, where the page table pointer is used by the operating system to point to the address space of the currently active process.

The approach of using indirection can be relatively inefficient if the memory resource that holds state in the guest VM has characteristics that are different from those on the native platform, as in the example of **Figure 8.3**a, where the register state is actually held in memory. In such a case, in order to perform a copy operation from register A of virtual machine 2 to register B of the same virtual ma-

chine, the VMM would have to perform a memory-to-memory copy by loading the value in the memory location representing register A into some temporary register and then storing the value in the memory location corresponding to register B. Such a copy takes two memory-access instructions and could take more instructions if an address register must be loaded with a pointer to the location of the register file in memory.

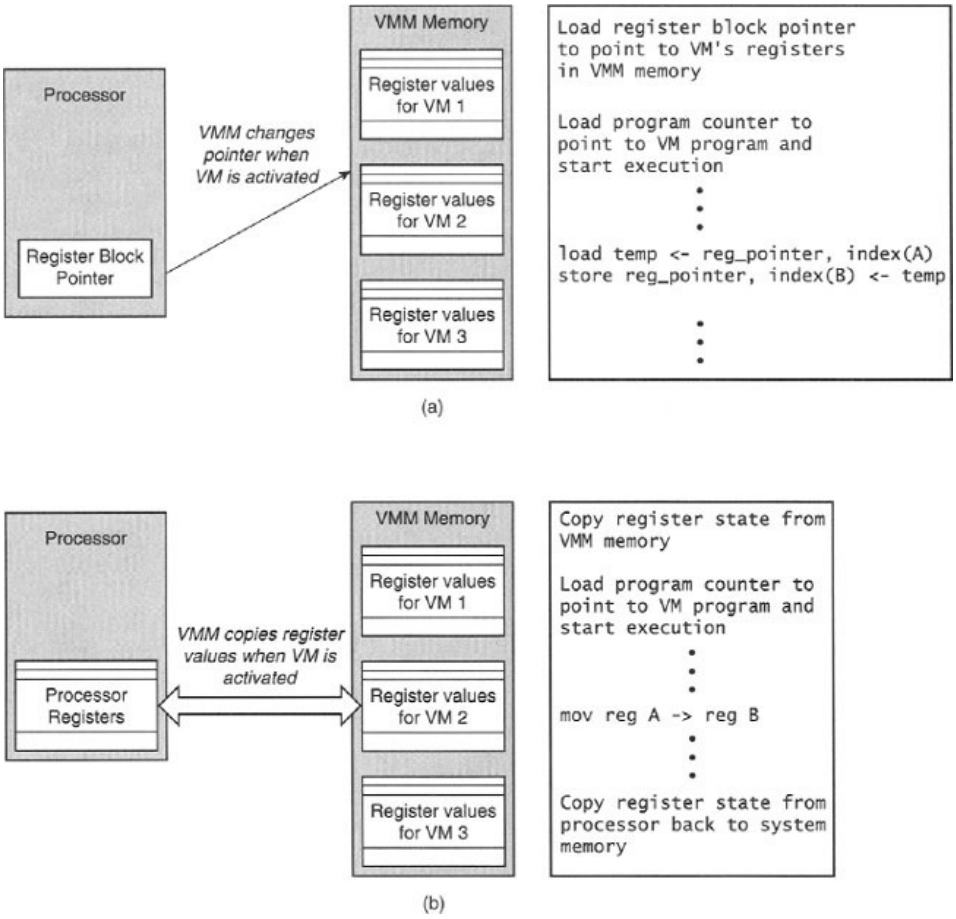


Figure 8.3 Methods of Maintaining Virtual State Illustrated for a Register File. *(a) Using indirection, (b) by copying. The boxes at the right indicate the actions taken when activating a virtual machine and copying a register value. In the first case, the value corresponding to a register is loaded from a table in memory and copied back to another location in the table. In the second case, the registers are used directly to accomplish the move.*

To avoid this inefficiency, a second approach for managing guest state is to copy a guest's state information to its natural level in the memory hierarchy whenever it is activated by the VMM and to copy it back when a different guest is activated. For example, in **Figure**

8.3b, the VMM copies the entire guest register contents into the host's register file at the time virtual machine 2 is activated by the VMM (after saving the registers of the previous guest). With this approach, the operation of moving a value from register A to register B is achieved by directly executing a native register-move instruction. In many system VM implementations, the goal is to execute guest code natively (i.e., without emulation) on the host platform. Thus, in the register example, after the one-time overhead of switching out the old virtual machine registers and switching in the registers of virtual machine 2, execution can proceed at the same rate as on a machine running the guest natively.

The choice between indirection and copying depends on the frequency of use and whether the guest state being managed by the VMM is held in a different type of hardware resource than on a native platform. For frequently used state information, such as the general-purpose registers, it is usually preferable to swap the state of the virtual machine to the corresponding real resource each time the virtual machine is activated.

8.1.3 Resource Control

Within a system VM environment, hardware resources, including the processor resource, are assigned to a VM at the time it is created according to configuration specifications. Once resources are given to a guest VM, it is important that there be a way for the VMM to get them back so that they can be assigned to a different VM. Thus, the VMM must maintain overall control of all the hardware resources, even though they are temporarily being used by the guest VM currently running. This section examines ways in which this can be achieved.

First note that a similar issue arises in the case of conventional time-sharing systems—multiple jobs are concurrently being executed on the machine, with each job having full access to its resources at any

given instant. However, there are some resources in the machine that are not directly accessible to the user application but accessible only to the operating system. One such resource is the interval timer, which, after being loaded with a value by the operating system, counts down clock ticks and triggers an interrupt when the value it holds becomes zero. Before passing control to a user process, the operating system initializes the timer with the maximum period of time it will allow the user process to run. The timer interrupt will then guarantee that control is transferred back to the operating system within the maximum time interval. Thus, the operating system uses the timer to keep control of the processor by ensuring that no user application can run for an indefinite length of time. And, because the OS has control of the processor, it can then control all the other resources. This control issue is similar to the one discussed in connection with runtime control of emulation frameworks in **Chapter 3**.

The situation is similar in a system VM environment. Here, the principal issue is that of time sharing resources among the different virtual machines. We will see in **Section 8.2.1** that a convenient way to provide the VMM with overall control is for the VMM to intercept all accesses to so-called privileged resources, such as the interval timer. These privileged resources, therefore, are not used directly by the individual virtual machines—rather, the VMM emulates the operation of these resources at all times. Instead of allowing the operating system in a virtual machine to field the timer interrupt, for example, the VMM first handles the interrupt itself. The handler for the timer interrupt includes code that saves the state of the current guest virtual machine, determines the next virtual machine that should be activated, and loads the state for the next virtual machine to be run. It accomplishes this, as mentioned earlier, either by changing pointers, by copying state into hardware resources, or by doing both. **Figure 8.4** depicts scheduling actions taken by the VMM.

There may be additional opportunities for the VMM to get control of the system other than through interval timer interrupts. For example, since privileged instructions encountered in the user mode must, as noted earlier, be intercepted and emulated by the VMM, the VMM gains control whenever the guest operating system issues privileged instructions (the mechanism for doing so is described in **Section 8.2**).

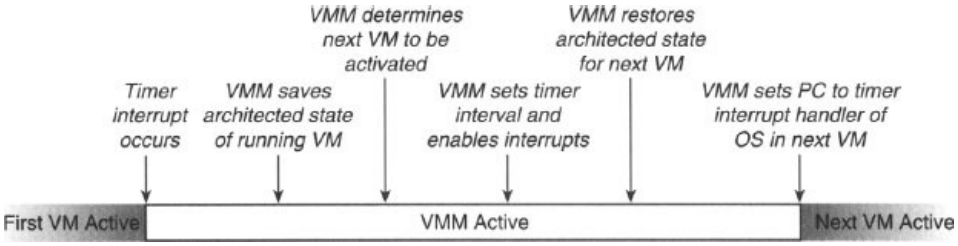


Figure 8.4 Actions Taken by the VMM in Retiring One Virtual Machine and Activating the Next Virtual Machine.

A fair scheme for allocating hardware resources to virtual machines is one in which the virtual machines get ownership of the resources in turn for roughly equal amounts of time. The issues here are similar to the issues that arise in multiprogrammed operating systems. If the time allocated to each virtual machine is large, there is the possibility that the resources are not optimally utilized. On the other hand, if the time allocated is small, the overhead incurred in switching between the virtual machines begins to degrade performance.

For a fair scheme to work, the guest operating system on a virtual machine must be denied direct access to the interval timer facility so that it cannot reschedule the next timer interrupt. Furthermore, in order to make operation under a virtual machine completely transparent, the guest operating system should not be allowed to read the real timer value set by the VMM. (This transparency is needed to prevent the operating system from behaving in a different manner when running under a VMM than it would on a real machine.) However, as we will see later in the chapter, the VMM can supply the guest with an emulated *virtual* interval timer.

Strict VMM emulation of a guest's timer interval could potentially lead some user code to run for too long a time, however, starving other virtual machines from using the hardware resources. Hence, the VMM must examine every guest setting of the timer interrupt and override the requested value if it has the potential of being unfair to other virtual machines that may be running on the system. This is an instance where the emulation of privileged resources must be done by the VMM with a view to efficient and fair operation of the entire virtual machine system rather than for the performance of an individual guest.

8.1.4 Native and Hosted Virtual Machines

It should be clear by now that the VMM is the key component in any system VM environment. The VMM is responsible for scheduling and managing the allocation of hardware resources to the various guest virtual machines. Hence it is also the point of control for the shared physical resources in the system. Such resources include the registers in the CPU, the real memory in the system, and the various I/O devices attached to the system. For *efficient* operation of the system, therefore, at least some part of the VMM implementation should have privileges higher than the actual privileges of the guest virtual machines it supports. By *actual privilege,* we mean the privilege the code has on the hardware while running in the virtual machine environment. This, as we will see shortly, may be different from the privilege that is *perceived* on the guest.

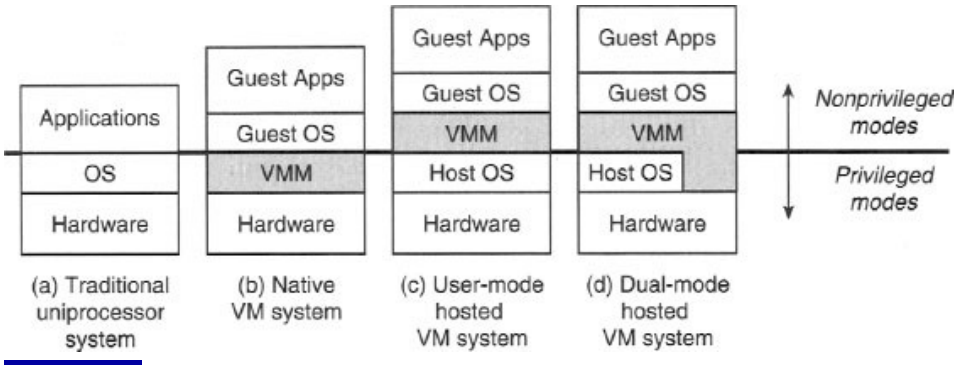


Figure 8.5 Native and Hosted VM Systems. *(a) The operating system of a traditional uniprocessor executes in the privileged mode. (b) In a native VM system, the VMM operates in the privileged mode. (c-d) A trusted host operating system executes in the privileged mode in a hosted VM system. In both systems, the guest operating system resides in the virtual machine and operates at a less privileged level. The VMM of a hosted system may work entirely in the user mode (c) or in dual mode (d) when parts of the VMM operate in the privileged mode.*

As pointed out earlier, the relationship between the VMM and virtual machines is analogous to the relationship between an operating system and application programs in a conventional time-shared system. In the latter, the operating system typically works in a privilege level higher than that of the applications, e.g., in *system mode* versus *user mode,* as shown in **Figure 8.5**a. A virtual machine system in which the VMM operates in a privilege mode higher than the mode of the guest virtual machines is called a *native VM* system. In a native VM, the VMM is the only software that executes in the highest privilege level defined by the system architecture, as depicted in **Figure 8.5**b. Conceptually, the VMM is first installed on the bare hardware, and the guest operating systems are then installed on top of the VMM. The guest operating systems and other less privileged applications run in levels of privilege lower than that of the VMM. This typically means that the privilege level of the guest OS is emulated by the VMM. Native VM systems have been used and studied extensively and are discussed in detail in upcoming sections.

For user convenience and implementation simplicity, it is often advantageous to install a virtual machine system on a host platform that is already running an existing OS. Such a system is called a *hosted VM* system (in this case the term *host* refers to the underlying OS). In a hosted VM system the VMM utilizes the functions already available on the host OS to control and manage resources desired by each of the virtual machines. Modifying a proprietary commercial operating system is often not possible, either because the source code is unavailable or because of licensing agreements. In these cases, the VMM may be implemented at the user level, a privilege level below that of the host operating system, as shown in **Figure 8.5**c. Such a system is commonly referred to as a *user-mode* hosted

VM system. For efficiency reasons, however, it is desirable to have at least part of the VMM work in the most privileged mode. This can often be achieved by modifying the host operating system through mechanisms commonly provided to extend the functionality of an operating system, such as kernel extensions or device drivers. Such a system that has parts of the VMM operating in a privileged mode and other parts in nonprivileged mode, as depicted in **Figure 8.5**d, is referred to as a *dual-mode* hosted VM system. We will examine features of such systems further in **Section 8.4.3**.

8.1.5 IBM VM/370

The first virtual machine environment was the IBM System/360 Model 40 VM (circa 1965) (**Adair et al. 1966**; **Creasy 1981**). The intention of the developers was to build a time-sharing system that extended the then-novel concept of virtual memory to other parts of the computer system. After debating the alternatives, the design team decided that the most appropriate interface to provide to the user would be the full System/360 architecture interface (the ISA) rather than just the user-level architectural interface enhanced by system library functions (the ABI). The perceived advantage was that this would be the best way to protect users from one another while preserving compatibility as the System/360 architecture evolved. It was not until much later, with a model of the System/370, that VM became mainstream. We will refer to the IBM virtual machine in the rest of the chapter as the IBM VM/370—many of the principles are still in use today in the z/VM running on the IBM zSeries mainframes. An informative account of the development of VM on the IBM platforms is provided in **Varian (1997)**, while a survey of virtual machine research at that time appears in **Goldberg (1974)**.

The virtual machine monitor of VM/370 was called the *control program* (CP). The CP design team also developed a single-user operating system, called the conversational monitor system (CMS), mainly

ing system called the *conversational monitor system* (CMS), mainly to demonstrate the advantages of modularization for system evolution. The CP/CMS design separated the function of resource management from the function of providing services to the user. The success of the project was in large measure due to the design of the System/370 architecture, which allowed an elegantly simple implementation of the virtual machine concept.

Even though CP and CMS are often referred to as a pair, one can exist without the other. In fact, CMS was developed on the bare machine before CP existed, and CP has hosted many operating systems in its life, including, most recently, Linux. As its importance and popularity grew, special hardware features were added to the System/370, primarily to reduce virtualization overhead. Some of these enhancements will be discussed in **Section 8.5**.

8.2 Resource Virtualization—Processors

The key aspect of virtualizing a processor lies in the execution of the guest instructions, including both system-level and user-level instructions. There are two ways this can be done. The first is through *emulation.* As described in **Chapter 2**, emulation can be performed through either interpretation or binary translation. Emulation involves examining each guest instruction in turn, either repeatedly when interpreting or once when performing binary translation, and emulating on virtualized resources the exact actions that would have been performed on real resources. Emulation is the only processor virtualization mechanism available when the ISA of the guest is different from the ISA of the host. As we shall see presently, however, an emulation-like process may be necessary at times even when the two ISAs are identical. Such a situation occurs when instructions that interact with hardware resources need to operate differently on a virtualized processor than on a real processor

differently on a virtualized processor than on a real processor.

The second processor virtualization method uses *direct native execution* on the host machine. This method is possible only if the ISA of the host is identical to the ISA of the guest, and even then only under certain conditions. It is always possible to build a virtual machine by using emulation. However, even with sophisticated techniques such as binary translation, the performance of a program on the virtual machine will rarely be as good as its performance on native hardware. Therefore, a basic design goal for system VMs with identical guest and host ISAs is that a significant fraction of the instructions execute directly on the native hardware. If this is possible, a program will often run on a virtual machine at about the same speed as on native hardware, unless there are memory or I/O resource limitations. The overhead of emulating any remaining instructions depends on several factors, including the actual number of instructions that must be emulated, the complexity of discovering these instructions, and the data structures and algorithms used for emulation.

The following three subsections characterize those instructions in an ISA that must be emulated in order for a VMM to be successfully constructed. The techniques used for emulating the special instructions depend on the characteristics of the ISA. We will see that for a well-behaved, efficiently virtualizable ISA, a trap occurs naturally when an instruction needs to be emulated, the trap handler jumps to an appropriate interpreter routine, interprets the single instruction, and returns control back to the original program. In other, less well-behaved ISAs, the difficulty of isolating the instructions that need to be emulated may force a larger number of other instructions to be emulated. **Section 8.2.4** addresses the problem of discovering such instructions. In these cases, emulation performance can be improved by translating a group consisting of several instructions at a time using techniques similar to those described in **Chapter 2**.

8.2.1 Conditions for ISA Virtualizability

In a classic paper (**Popek and Goldberg 1974**) inspired by Goldberg's doctoral dissertation (**Goldberg 1972**), Popek and Goldberg formally derived sufficient conditions under which an ISA can efficiently support virtual machines. We summarize in this subsection the valuable insight regarding the construction and understanding of system VMs provided by this work.

In **Section 8.1.4** we characterized both native and hosted system VMs. There tends to be an inherent inefficiency in purely hosted VMs because, as we will see in **Section 8.4.3**, the VMM needs to rely heavily on services of the host operating system. Therefore, we restrict the discussion here to native system VMs. In a native system VM, the VMM runs in system mode, and all other software runs in user mode. Note that this "other" software includes some software originally intended to run in the system mode, a guest operating system being the prime example. The VMM keeps track of the intended (i.e., virtual) mode of operation of a guest virtual machine but sets the actual native hardware mode to *user* mode whenever executing instructions from the guest virtual machine, regardless of whether it is user or system software.

The original analysis by Popek and Goldberg was for "third-generation" machines, such as the IBM System/370, the Honeywell 6000, and the Digital PDP-10, but it still holds for present-day machines (one could argue that from an ISA perspective, we are still in the third generation!). The assumptions made in the analysis are as follows: (a) The hardware consists of a processor and a uniformly addressable memory, (b) the processor can operate in one of two modes, the system mode or the user mode, (c) some subset of the instruction set is available only in the system mode, and (d) memory addressing is done relative to the contents of a relocation register (i.e., a fixed value is added to a virtual address to arrive at a physical