memory address). The analysis could be extended to more than two modes and to more complex memory architectures, but the fundamental results remain unchanged. Input/Output is not considered, but the approach and analysis can also be extended to I/O in a straightforward manner.

The machine being virtualized is modeled as a 4-tuple, S = ≤ *E, M, P, R*>, where *E* refers to the executable storage, *M* refers to the mode of operation, *P* is the program counter, and *R* is the memory relocation bounds register. The memory of the machine is assumed to be located contiguously in physical memory from the location given in the relocation bounds register *R*. The relocation bounds register, as the name suggests, is a pair indicating the physical location and the size of the virtual memory space. The extension to paged or segmented virtual memory systems is straightforward. A *memory trap* occurs if the address accessed by a program falls outside the bounds indicated by *R*. The trap atomically saves the current state of the machine, represented by *M, P,* and *R* in location 0, and copies the contents of location 0 of another process into *M, P,* and *R*—this essentially has the effect of performing a context switch.

A *privileged* instruction is defined as one that traps if the machine is in user mode and does not trap if the machine is in system mode. It is not sufficient for the instruction's behavior to be different in the two modes, and by definition the privileged instruction *must* trap when executed in the user mode. Note that this strict definition is for the purpose of this particular analysis and may differ from definitions used elsewhere when describing specific ISAs. Here are two examples of privileged instructions.

- **Load PSW** (LPSW, IBM System/370): This instruction loads the processor status word (PSW) with a doubleword from a location in memory if the processor is in system mode. If it is not in system mode, the machine traps. The PSW contains bits that determine, among other things, the state of the CPU. One of these bits, the P bit, specifies whether the CPU is in user or system mode.

Another part of the PSW is the instruction address (or program counter). If this instruction could be executed in user mode, a malicious user program could put itself in system mode and get control of the system.

- **Set CPU Timer** (SPT, IBM System/370): This instruction replaces the CPU interval timer with the contents of a location in memory if the CPU is in system mode and traps if it is not. Once again, if this instruction could be executed in user mode, it would be possible for a user program to change the amount of time allocated to it before being swapped out.

As noted earlier, in a virtual machine environment, an operating system running on a guest virtual machine should not be allowed to change hardware resources in a way that affects the other virtual machines or the performance of programs outside the realm of its own virtual machine. Hence, even the operating system on a virtual machine must execute in a mode that disallows the direct modification of system resources such as the PSW and the CPU interval timer. Consequently, all of the guest operating system software is forced to execute in user mode.

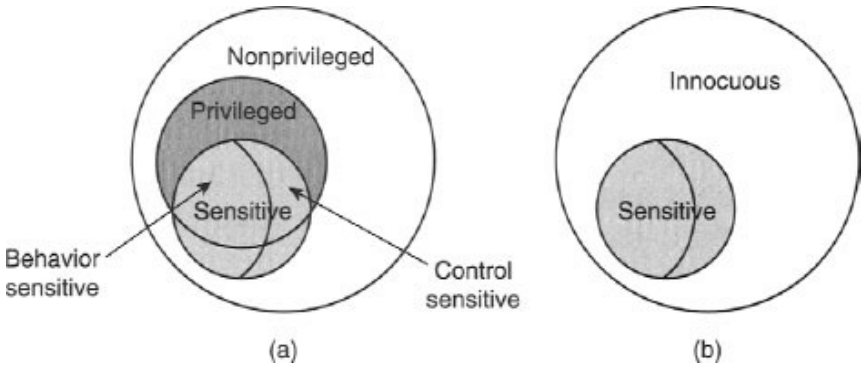


Figure 8.6 Types of Instructions. *(a) Sensitive and privileged instructions overlap (although not necessarily completely). (b) Sensitive instructions and innocuous instructions are complements of each other.*

To specify instructions that interact with hardware, two categories of special instructions are defined. *Control-sensitive* instructions are those that attempt to change the configuration of resources in the system, for example, the physical memory assigned to a program or the mode of the system. (If I/O were included in the model, then I/O would also be considered a resource.) *Behavior-sensitive* instructions are those whose behavior or results produced depend on the configuration of resources—in the model, this includes the value in

the relocation bounds register or on the mode of operation. If an instruction is neither control sensitive nor behavior sensitive, it is termed *innocuous* (see **Figure 8.6**).

The LPSW and SPT instructions mentioned earlier are examples of control-sensitive instructions. Their use allows an operating system to change some basic resource of the system, such as the mode of operation or the CPU timer. Examples of behavior-sensitive instructions follow.

- **Load Real Address** (LRA, System/370): This instruction takes a virtual address, translates it, and saves the corresponding real address in a specified general-purpose register. The behavior of this instruction (i.e., the result value stored in the register) depends on the state (mapping) of the real memory resource.
- **Pop Stack into Flags Register** (POPF, Intel IA-32): This instruction pops the flag registers from a stack held in memory. One of the flag registers is the interrupt-enable flag, which can be modified only in privileged mode. In user mode, this instruction overwrites all flags except the interrupt-enable flag. For the interrupt-enable flag, the instruction acts as a no-op when executed in user mode. The behavior of this instruction therefore depends on the mode of operation.

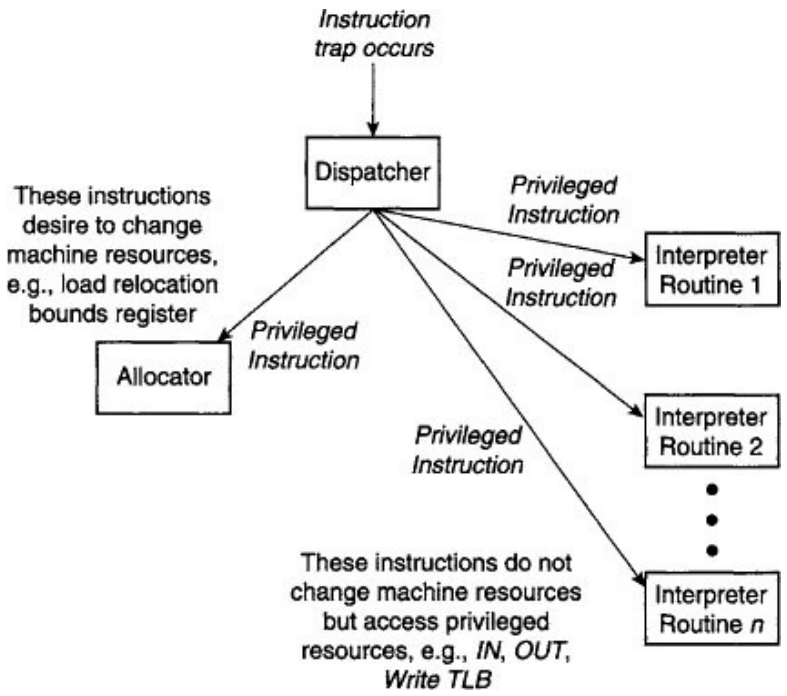


Figure 8.7 Components of a Virtual Machine Monitor.

The functions of the VMM can be divided into three parts, a *dispatcher*, an *allocator*, and a set of *interpreter routines*. These components of the VMM are depicted in **Figure 8.7**. The dispatcher is the top-level control module of the VMM, which decides the next module to be invoked; it is invoked by the interrupt handler when the hardware traps. The allocator decides how system resources are to be allocated, e.g., how to allocate memory resources in a nonconflicting manner. The allocator is invoked by the dispatcher whenever there is a need to change machine resources associated with some virtual machine.

As we shall see, in a well-constructed VMM, any instructions that attempt to change resource assignments or whose behavior is affected by the assignment of resources will trap to the VMM dispatcher. The trapping instructions that attempt to change resource assignments are then directed by the dispatcher to the allocator. The dispatcher directs all remaining traps to the interpreter routines. The interpreter routines, one per privileged instruction, emulate the effects of the instructions when operating on virtual resources. After an interpreter routine finishes, control is passed back to the guest virtual machine at the instruction immediately following the one that caused the trap.

According to Popek and Goldberg, a potential virtual machine monitor must satisfy three properties before qualifying as a true virtual machine monitor: efficiency, resource control, and equivalence. *Efficiency* implies that all instructions that are innocuous must be executed natively on the hardware, with no intervention or emulation by the VMM. *Resource control* implies that it should not be possible for guest software to directly change the configuration of any system resources available to it, e.g., real memory. The allocator must be invoked if the guest software makes any such attempt. *Equivalence* implies that any program executing on a virtual ma-

chine must behave in a manner identical to the way it would have behaved when running directly on the native hardware, with only a few exceptions. The allowed exceptions are that (1) performance can be slower due to the emulation (e.g., interpretation) of certain instructions, (2) there may be a limitation on the extent of resources available, e.g., disk space, because of sharing among virtual machines, and (3) there may be differences in performance due to changed timing relationships, e.g., between the I/O and the processor.

Here, the reader should note that this formulation of a VMM is somewhat at odds with the common definition and the definition used elsewhere in this book. It is intuitively clear that the resource control and equivalence properties should hold for a VMM. Furthermore, it should always be possible to construct a VMM with these properties, however inefficient, by employing emulation techniques whenever necessary. In effect, it is these two properties that we (and others) typically assume when defining a VMM. In the Popek and Goldberg formulation, a VMM must also be “efficient” in order to be considered a true VMM. And it is the efficiency condition that most heavily depends on the condition stated shortly ahead in Theorem 1.

An alternative formulation would define a VMM to have only the properties of resource control and equivalence and then to define an *efficient VMM* to be a VMM that also satisfies the efficiency property. With the exception of the Popek and Goldberg theorems, we use these alternative definitions of VMMs and efficient VMMs in this book. Here, then, is the key theorem regarding (efficient) VMMs.

Theorem 1: For any conventional third-generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

This is a surprisingly simple theorem, yet it has important implications. It says that an efficient virtual machine implementation can be constructed if instructions that could interfere with the correct or efficient functioning of the VMM (i.e., sensitive instructions) always trap in the user mode. All nonprivileged instructions can be executed natively on the host platform; no emulation is needed. The trapping of sensitive instructions, such as ones that attempt to reallocate system resources or depend on system resource assignments, force control to go back to the VMM. The VMM can then examine the action desired by the virtual machine that issued the sensitive instruction and reformulate the request in the context of the virtual machine system as a whole. **Figure 8.8** illustrates the sufficient condition contained in the theorem.

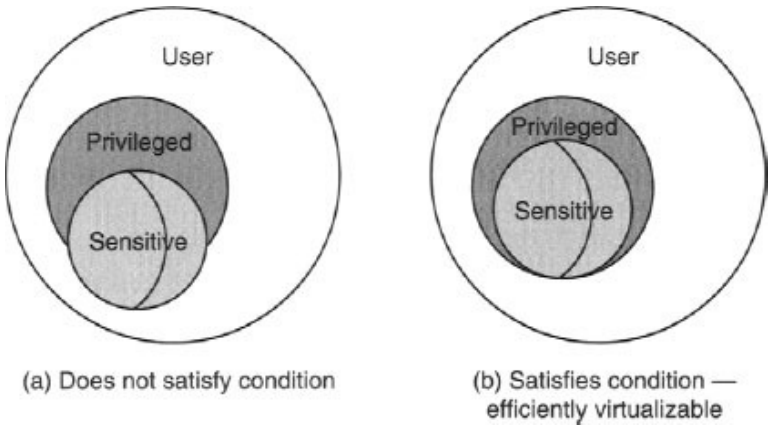


Figure 8.8 Illustrating Popek and Goldberg's Theorem 1. *In (a), the sensitive instructions are not a subset of the privileged instructions and hence the system may not be efficiently virtualizable. On the other hand, the system in (b) satisfies the condition of Theorem 1.*

The VMM interprets a sensitive instruction according to the prevailing status of the virtual system resources and the state of the virtual machine. Take, for example, the sensitive instructions LPSW and SPT described earlier in this section. These System/370 instructions are also privileged and hence satisfy the condition indicated in Theorem 1. Let's assume that a particular instance of an LPSW instruction has the net effect of going from one privileged state to another state with the same set of privileges. The LPSW interpreter routine in the VMM determines that this particular transfer is harm-

less by examining and comparing the contents of the location being loaded with the contents of the virtual copy of the PSW it maintains for that virtual machine. It then transfers control appropriately by modifying the contents of the hardware PSW, using an LPSW of its own. This LPSW does not trap because the VMM operates in system mode. **Figure 8.9** illustrates this process.

On the other hand, if the LPSW instruction were executed by a user application running on the virtual machine, the trap handler would generate a virtual trap that is then passed to the virtual machine. The guest operating system handles this trap in exactly the same way as a native machine would when an attempt is made to execute a privileged instruction in user mode.

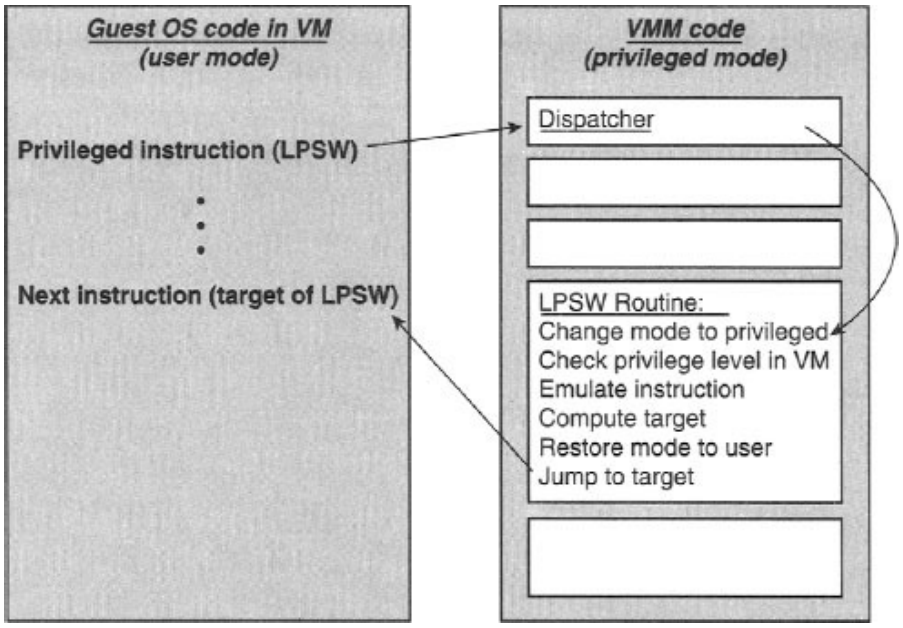


Figure 8.9 Handling of a Privileged Instruction in a Guest OS. *The privileged instruction causes a trap because the guest OS is operating in user mode. The trap goes to the dispatcher in the VMM, from where it is directed to the special interpreter routine for the specific instruction.*

When interpreting the SPT instruction, the VMM examines the contents of the location to be loaded into the CPU timer. It will allow this value to be loaded if the value (say, t) is less than the time remaining from the allocated time for the virtual machine itself ($t \leq T$). Otherwise it will load the timer with the time remaining for the vir-

tual machine (*T*). Meanwhile, it keeps the time difference (*t*—*T*) in an internal table so that this time can be restored when the guest VM is again activated. In this manner, the VMM ensures that it always remains in control, even in the presence of attempts by some virtual machine to monopolize the system resources. If the guest OS should attempt to read the timer value, the VMM will reconstruct the correct virtual timer value and return it to the guest.

There are several simplifying assumptions made concerning the types of architectures under consideration in developing Theorem 1, and some practical aspects such as I/O and asynchronous interrupts have been ignored. Yet the theorem is useful because it suggests a convenient way of testing an ISA to determine the feasibility of an efficient virtual machine monitor. The theorem also provides clear guidelines for computer architects wishing to ensure that a new ISA lends itself to efficient virtualization.

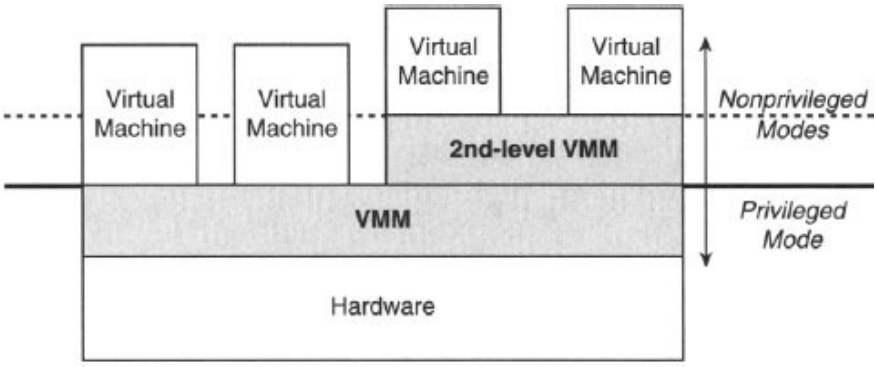


Figure 8.10 Recursive Virtualization.

8.2.2 Recursive Virtualization

It may sometimes be desirable to run a virtual machine system as one of the virtual machines. This implies that the VMM itself would be running in the user mode under the control of a copy of itself running in the privileged mode. The concept of running the virtual machine system on a copy of itself is called *recursive virtualization* (**Figure 8.10**).

There are two effects that usually restrict the ability to create an efficient recursively virtualizable system. First, if the VMM itself has timing dependences, then its performance could be affected adversely when it runs in user mode within another virtual machine. In fact, the presence of timing dependences in a program could end up violating one of the conditions of Popek and Goldberg—the equivalence property. Thus a strong requirement for a recursively virtualizable system is that a VMM without any timing dependences be constructible for the system.

The second effect comes from the fact that each VMM layer uses up its own resources, particularly system memory resources needed to keep the state of the various virtual machines running on the system. If recursive virtualization is repeatedly performed, then these resources keep shrinking to a point where little is left for allocation to the virtual machines. As a result, the number of programs that run with identical effect on the real and virtual machines shrinks as the number of levels of virtualization increases.

Neither of these is a problem, in practice, however. Seldom does the required number of levels of virtualization go beyond two. At this level of recursion, the overhead is still contained; moreover, the user is usually willing to accept the drop in performance resulting from the recursive virtualization.

Popek and Goldberg formalized the conditions for recursive virtualizability in a second theorem.

Theorem 2: A conventional third-generation computer is recursively virtualizable if (a) it is virtualizable and (b) a VMM without any timing dependences can be constructed for it.

8.2.3 Handling Problem Instructions

Observe that, as described earlier, the POPF instruction in the Intel IA-32 ISA is sensitive but not privileged—it does not generate a trap in user mode. Hence it violates the virtualizability condition of Theorem 1. The Intel IA-32 has several other instructions that similarly are sensitive but not privileged (**Robin and Irvine 2000**). In reality, there are very few ISAs that are virtualizable in the sense of the Popek-Goldberg theorem (or efficiently virtualizable in the terminology we use). However, if an ISA is not efficiently virtualizable, all is not lost. It simply means that an additional set of steps must be taken in order to implement a system virtual machine (with possible loss of some efficiency).

The POPF instruction mentioned earlier is a problem instruction because it inhibits the creation of an (efficient) virtual machine monitor in the sense of Theorem 1. For convenience, let us refer to instructions that are sensitive but not privileged as *critical* instructions. It would be possible for a VMM to intercept POPF and other critical instructions if all guest software were interpreted instruction by instruction. The use of interpretation clearly leads to inefficiency (both as formally defined by Popek and Goldberg and in reality). Fortunately, techniques related to those described in **Chapters 2** and **3** can be used to reduce the inefficiency considerably. For example, the VMM can scan the guest code stream before execution, as done during binary translation, and in the process discover all critical instructions, replacing them with a trap or a jump to the VMM, as shown in **Figure 8.11**. This replacement process is known as *patching* (**Section 4.7.1**). Only the code running on the guest virtual machines needs to be scanned—the VMM code is in privileged state and will work correctly without modification. The problem of scanning and patching critical instructions is discussed in more detail in the next subsection.

In a strictly efficient virtualizable machine, as defined by Popek and Goldberg, all nonprivileged instructions are executed natively. We refer to a virtual machine system in which some of the nonprivi-

leged instructions must be emulated as a *hybrid virtual machine* system. Scanning and patching is a pragmatic attempt by a hybrid virtual machine system to execute *most* nonprivileged instructions natively. Guest VM software containing few critical instructions should not see a significant degradation in performance.

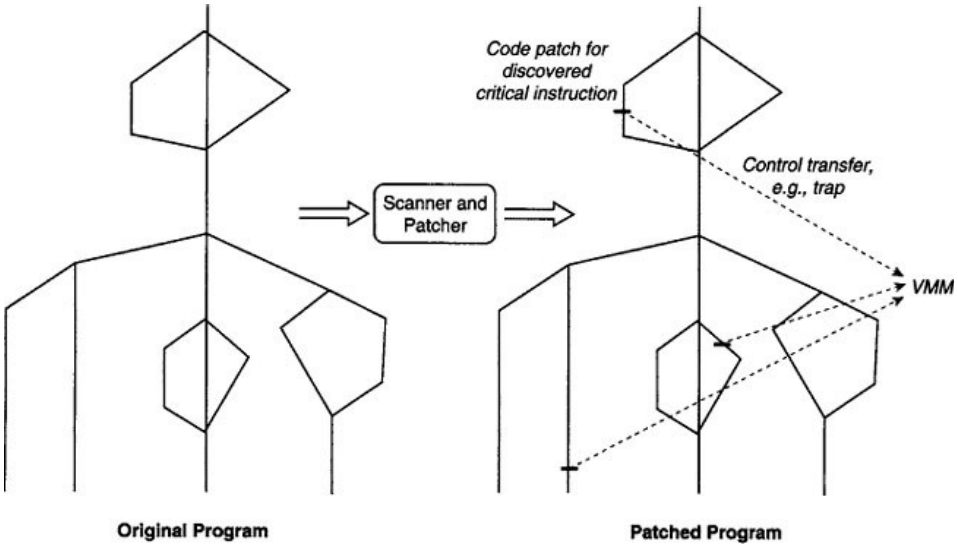


Figure 8.11 Scanning and Patching Code in a Hybrid Virtual Machine System.

8.2.4 Patching of Critical Instructions

As noted earlier, the problem of discovering critical instructions has a lot in common with the general problem of dynamic binary translation discussed in **Chapter 2**. The basic idea in both cases is to locate sections of code about to be executed and perform transformations on them. A major difference is that in binary translation, both the target code and the runtime execute in user mode, and hence a switch between them can be done via a jump. In the VM context, there is an actual change of privilege mode, which means that the jumps to the runtime are replaced with system calls (traps) to the VMM.

One way to discover critical instructions is for the VMM to take control at the head of each guest basic block and scan instructions in se-

quence until the end of the basic block is reached. If a critical instruction is found during this scan, it is replaced with a trap (system call) to the VMM, and the critical instruction at the location is saved in a VMM side table, along with its original address. Another trap back to the VMM is placed at the end of the basic block in order to allow the VMM to regain control when the block completes execution so that it can scan and patch the next basic block to be executed. If the next block has already been patched, the VMM simply jumps to the next block. This method ensures that every basic block to be executed is patched first.

To reduce overhead, the trap at the end of a scanned basic block can be replaced by the original branch or jump instruction after all possible successor basic blocks have been patched; this is analogous to chaining basic blocks in a code cache (**Section 2.7.1**). For example, if the basic block ends with a conditional branch that has been replaced with a trap, the basic blocks located at both the taken and the fall-through addresses can be scanned the first time the trap is encountered. After patching any critical instructions contained in the successor blocks, the trap is replaced with the original conditional branch. This technique is relatively simple with direct branches, i.e., branches that have a target specified as an offset relative to the address of the branch instruction itself. For indirect jumps and branches where the destination address is located in a register, the trap typically cannot be replaced, because of the difficulty in predicting all possible target addresses.

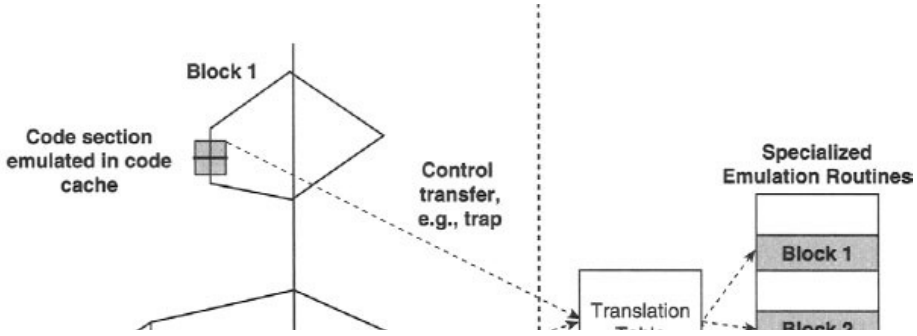
In order to further reduce overhead, instead of scanning single basic blocks at a time, the VMM can scan through all branches that allow target computation. Thus for a conditional branch, both the fall-through as well as the taken branch paths are followed during the initial scan, and this process is repeated until a control-transfer instruction is encountered for which it is not possible to determine all targets. In order to avoid memory paging caused by scanning possibly unused paths, the scanning should be limited to those code

pages already in physical memory.

Observe that a critical instruction appearing in a guest application program (i.e., in the VM's user mode) needs to be patched only if it causes some action to be taken when performed in user mode. For example, if a critical instruction behaves like a no-op in user mode, and if it can be determined that the instruction is to be executed by its VM in user mode, it is not necessary to patch the instruction with a trap. Thus, some of the overhead of trapping can be reduced in the virtual user mode. Unfortunately, as long as the instruction set contains even one critical instruction that, in user mode, performs either a read or a write action on some resource, the instruction must be emulated and the overhead of code scanning and patching must be incurred. Finally, the process of scanning and patching critical instructions is often sensitive to the characteristics of the underlying ISA and is fertile ground for engineering optimizations beyond those described here.

8.2.5 Caching Emulation Code

The overhead of VMM interpretation can become a problem when the frequency of sensitive instructions requiring interpretation is relatively high. One way to reduce this overhead is to cache the actions taken during the first interpretation, with a view to avoiding many of the interpretation steps when the same instruction is encountered in the future. This is similar to code caching as described in connection with process virtual machines in **Chapter 3**.



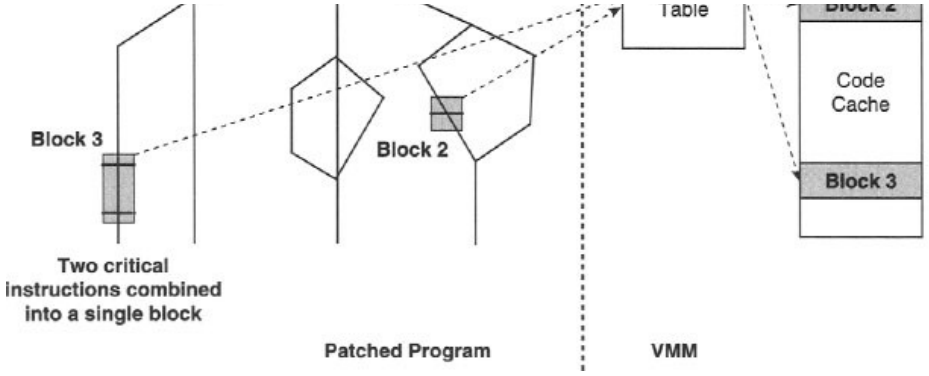


Figure 8.12 Code Caching to Improve Performance of Emulating Critical Instructions.

Ideally, code caching is done on a block of instructions surrounding the sensitive instruction, since larger blocks lend themselves better to optimization. In contrast to what was shown in **Figure 8.11**, a system call trap may be inserted at an instruction earlier than the critical instruction. As depicted in **Figure 8.12**, the trap causes control to be transferred to the VMM, which then locates the cached emulation code via a table lookup using the address of the trapping instruction. This cached code is an emulation of an entire block of the original code, including the critical instruction, and hence is specific to the location of the trapping instruction. When the VMM returns control to the virtual machine, it restores the program counter to the point immediately after the block that was emulated rather than to the instruction following the trap.

Unlike the simple interpretation case, where there is only one copy of the code that emulates all instances of a specific type of instruction, each instance of the critical instruction in the application will be associated with a distinct piece of cached code. The advantage here is that the cached code can be made specific to the instance and hence can be optimized, often in a way that eliminates a significant fraction of the overhead.

In contrast to process virtual machines, the management of this code cache is somewhat simpler, because code in the cache is executed after the trap, when the machine is already in system mode. The user code can never corrupt or read the contents of the code

The user code can never corrupt or read the contents of the code cache, which is therefore always protected from the user application. On the other hand, self-modifying code remains a problem, and mechanisms are needed to intercept modifications of the original code—such modifications may necessitate invalidations of parts of the code cache.

8.2.6 Efficient Virtualizability of Common Instruction Sets

The System/370 architecture provides two modes of operation, a *user* mode and a *privileged* mode (what we have been referring to generally as *system mode*). The operating system is expected to run in the privileged mode. Some of the instructions in the instruction set are privileged; they perform the specified action only when the processor is in privileged mode. These privileged instructions typically manage protected resources in the processors. In order to prevent either malicious or accidental modification of these resources, the privileged instructions trap when executed in the user state.

The VMM for System/370 (called the control program, or CP) has to run in the privileged mode in order to be able to allocate resources to different virtual machines at different times. The virtual machine is created in the privileged mode, but it executes in the user mode. This ensures that each virtual machine is isolated from other virtual machines and from the CP itself.

All sensitive instructions on the System/370, in the Popek-Goldberg sense, are also privileged. VM/370 is an example of a virtual machine employing direct native execution.

As noted earlier, an interesting aspect of the System/370 VM project was the development of a single-user operating system that proceeded in parallel with CP development. This operating system, called the Conversational Monitor System (CMS), took advantage of

the CP environment by allowing all applications to run in the "pseudo-supervisor," state with the user being allowed to use any instruction, whether privileged or not. This is safe because CMS and applications execute on a virtual machine protected by the CP from wantonly affecting the real resources of the computer. The ability to use privileged instructions in their code allows users to experiment in many ways, e.g., to add input/output devices not allowed by the standard system. **Figure 8.13** shows a comparison of the handling of privileged instructions at various levels in various schemes. This feature of the CP/CMS system is highlighted in **Figure 8.13**c.

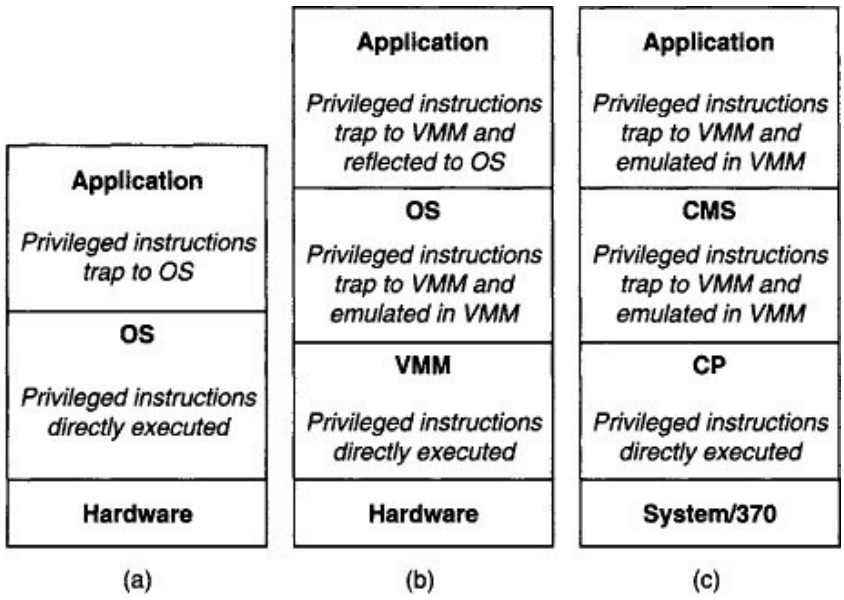


Figure 8.13 Comparison of the Execution of Privileged Mode Instructions on a VM/370 with that on a Native System and Other Virtual Machine Systems. *(a) A standard system; (b) a virtual machine system; (c) the CP/CMS system.*

Despite the fact that most of the concepts in this chapter have been known for at least 30 years, many ISAs developed during those 30 years do not satisfy the sufficient conditions of Theorem 1. For example, the most commonly used ISA today is the Intel IA-32, and the IA-32 contains 17 critical instructions (**Robin and Irvine 2000**). The likely reason for this is that system VMs have not been considered important because of the tremendous decrease in the cost of computer systems, which makes it easier for each user to have his/her own machine. However, in recent years there has been resurgence

of interest in VMs because of the importance of security in a networked environment and the desire to provide cross-platform compatibility of major software packages. Furthermore, large, expensive server systems are shared among many groups of users, and so several of the original VM motivations remain.

8.3 Resource Virtualization—Memory

In some respects, a virtual machine can be considered a generalization of the concept of virtual memory that preceded it. Virtual memory makes a distinction between the logical view of memory as seen by an application programmer and the actual hardware memory resource as managed by the operating system. Appendix A presents a discussion of the basic concepts underlying virtual memory. Also included is a discussion of the structures, e.g., page tables, needed to support virtual memory. It turns out that this hardware support is also sufficient for providing each guest virtual machine in a virtual machine system, with the view of having (and managing) its own "real" memory, which, in fact, is an illusion created by the underlying VMM.

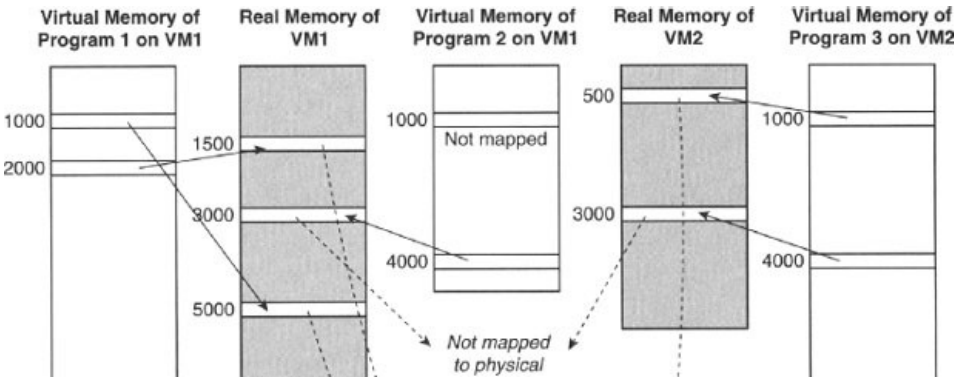
8.3.1 Virtual Memory Support in a System Virtual Machine Environment

In a system VM environment, each of the guest VMs has its own set of virtual memory tables. Address translation in each of the guest VMs transforms addresses in its *virtual* address space to locations in *real* memory—this real memory would correspond to the *physical* memory on a native platform; in a VM platform, however, this is not the case. Rather, in a system VM environment, a guest's real memory address must undergo a further mapping to determine the address in physical memory of the host hardware. Note that here we

are drawing a clear distinction between *real* memory and *physical* memory—terms often used interchangeably in other contexts. Physical memory is the hardware memory. Real memory is a guest VM's illusion of physical memory; an illusion supported by the VMM when it maps a guest's real memory to physical memory. It is this real-to-physical mapping that implements the *virtualization* of memory in a VM system.

The addition of another level in the memory hierarchy necessitates additional mechanisms for memory management. Note that the combined total size of the real memory of all the guests could be (and often is) bigger than the actual physical memory on the system. Hence the VMM maintains a swap space, distinct from the swap spaces of each of the guests. The VMM manages physical memory by swapping guest real pages into and out of its own swap space.

Figure 8.14 shows a set of page table maps belonging to two different guest VMs in a virtual machine system. Each entry in a *page table* maps a location in the virtual memory of an application to a location in the guest VM's real memory. In order to convert a real page address to a physical address, the VMM maintains a *real map table* mapping the real pages to physical pages. In the figure, the physical page frames numbered 500, 1000, and 3000 are assigned to two real pages of VM1 and one real page of VM2. The remaining physical pages may be allocated either to other virtual machines or to the VMM itself.



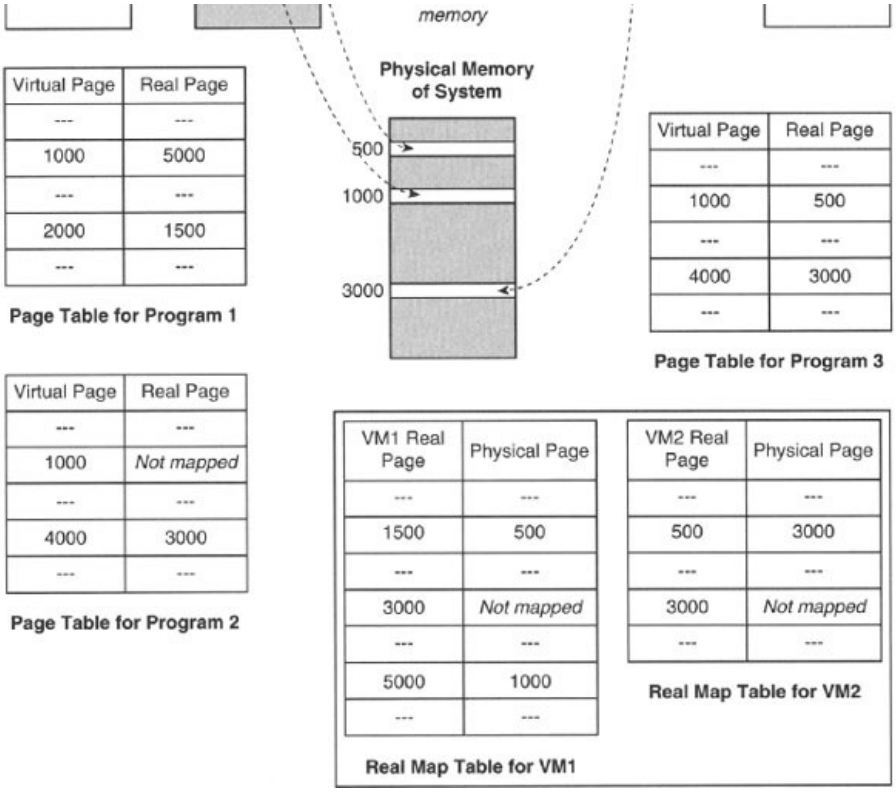


Figure 8.14 Memory Mapping of Guest VMs. *Virtual pages are mapped to real pages by page tables maintained by each virtual machine. Real pages are mapped to physical pages, using tables maintained by the VMM (at the bottom of the figure). Observe that the real memory of each VM can be larger than the physical memory—the real memory of VM1, for example, is as large as its virtual memory.*

On contemporary platforms, page translation is supported by a combination of a page table and a translation lookaside buffer (TLB); see Appendix Section A3.4. Depending on the ISA, either the page table or the TLB is architected. If the page table is architected, then its structure is defined by the ISA, and the operating system and hardware cooperate in maintaining and using it. In this case, the TLB is maintained and used only by the hardware; it is not visible to the operating system. When there is a TLB miss, the hardware "walks" the architected page table to find the appropriate entry to place in the TLB; if the entry is not mapped to physical memory, there is a page fault and the operating system takes over.

On the other hand, if the TLB is architected, then its structure and special instructions to manipulate it are part of the ISA; the page tables are part of the operating system implementation. Hardware is

unaware of the page table structure. With this approach, when a TLB miss occurs, there is immediately a trap to the operating system; the operating system then uses its page table information to perform the appropriate action.

Most of the older ISAs, including the Intel IA-32 and IBM System/370, use architected page tables; some of the more recent RISC ISAs use architected TLBs. Memory resource virtualization in a VM environment is done somewhat differently, depending on whether the page table or the TLB is architected. Because the most common system VMs in use today run on the Intel IA-32 or the IBM zSeries that evolved from System/370, we first consider virtualization with an architected page table. More information about virtual storage concepts in IBM VMs can be found in **Parmalee et al. (1972)**.

8.3.2 Virtualizing Architected Page Tables

The OS in each guest VM maintains its own page tables. These tables reflect the virtual-to-real memory mapping that the guest OS manages. As opposed to this virtual-to-real mapping, the virtual-to-physical mapping is kept by the VMM in *shadow page tables,* one for each of the guest VMs. **Figure 8.15** illustrates the shadow page tables for the example of **Figure 8.14**. These tables are the ones actually used by hardware to translate virtual addresses and to keep the TLB up-to-date. The entries in these shadow page tables essentially eliminate one level of indirection in the virtual-to-real-to-physical mapping.

To make this method work, the page table pointer register is virtualized. The VMM manages the real page table pointer and has access to the virtual version of the register associated with each guest VM. At the time the VMM activates a guest VM, it updates the page table pointer so that it indicates the correct shadow version of the guest's current page table. If a guest attempts to access the page table pointer, either to read it or write it, the read or write instruction

traps to the VMM. The trap occurs either automatically because these instructions are privileged, or because code patching has replaced them with a trap. If the attempt to access the page table pointer by the guest is a read attempt, the VMM returns the guest's virtual page table pointer; whereas if it is a write attempt, the VMM updates the virtual version and then updates the real page table pointer to point to the corresponding shadow table.

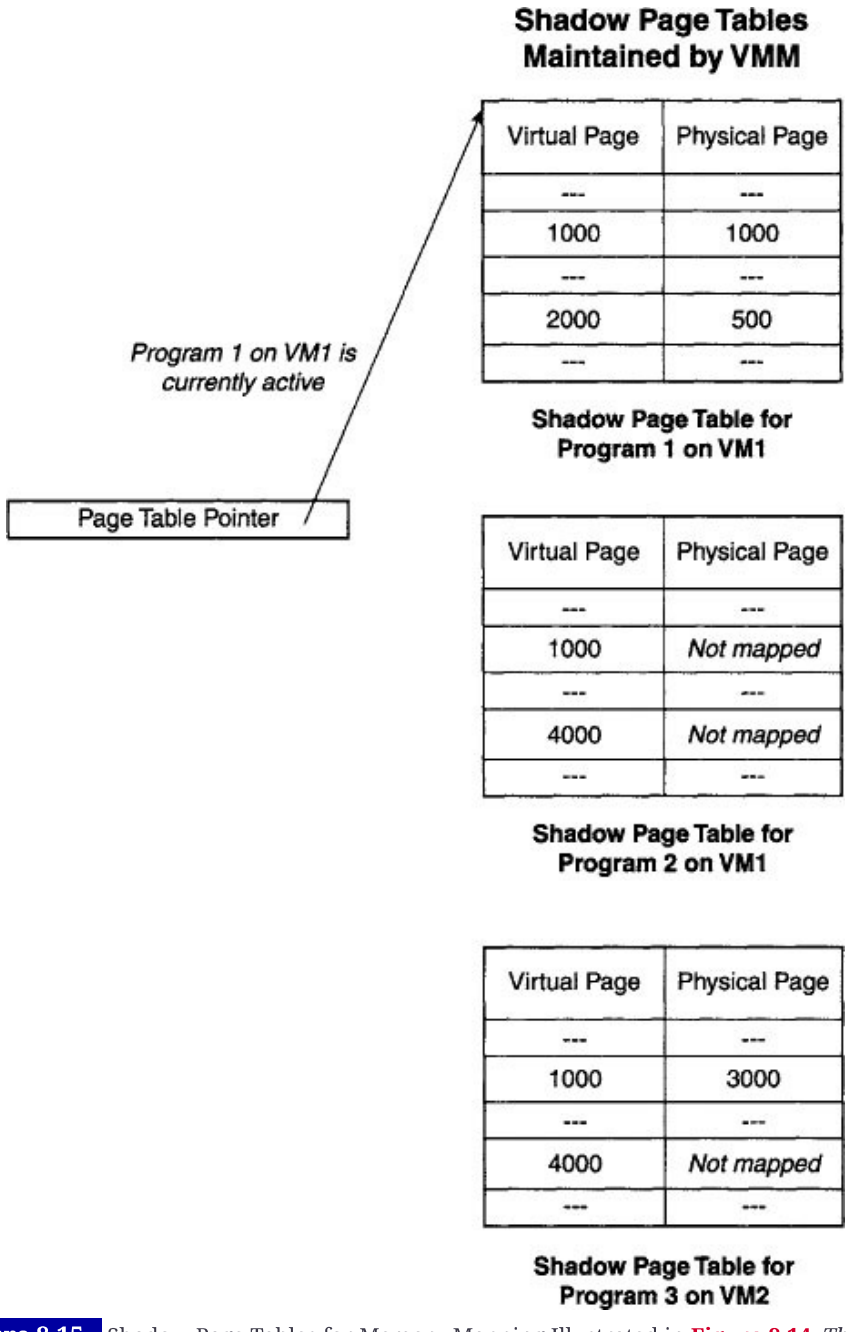


Figure 8.15 Shadow Page Tables for Memory Mapping Illustrated in **Figure 8.14**. *The shadow tables are used by hardware in performing address translation. The VMM manages the shadow tables and is responsible for setting the page table pointer register.*

The true mapping of virtual to physical pages may differ from the virtual to real view that the guest operating systems have, and page fault handling must take this into account. First note that the VMM should not have a virtual-to-physical page mapping in a shadow table if the guest OS does not have the same virtual page mapped to real memory in its corresponding virtual table. Otherwise, an access that should page fault from the guest's perspective will not cause a page fault in the VM environment, thereby breaking the equivalence property (compatibility). Therefore, when a page fault does occur, the page may or may not be mapped in the virtual table of the guest OS. If it is mapped, this page fault should be handled entirely by the VMM. This is a case where the VMM has moved the accessed real page to its own swap space. Consequently, the VMM brings the real page back into physical memory and then updates the real map table and the affected shadow table(s) appropriately to reflect the new mapping. The guest OS is not informed of the page fault because such a page fault would not have occurred if the guest OS were running natively.

On the other hand, if the page is not mapped in the guest, the VMM transfers control to the trap handler of the guest, indicating a page fault. The guest OS then issues I/O requests to effect a page-in operation (possibly with a swap out of a dirty page). The guest OS then issues instructions to modify its page table. These requests are intercepted by the VMM, either because they are privileged instructions or because the VMM write-protects the area of memory holding the page table of the guest. At that point the VMM updates the page table and also updates the mapping in the appropriate shadow page table before returning control back to the guest virtual machine.

As we indicated earlier, the real map table contains a mapping of the real pages of each virtual machine to the physical pages of the

system. When performing I/O with real addresses (as many systems do, particularly for paging operations), the VMM converts the real addresses presented by a virtual machine to physical addresses using the real map table. Input/output address mapping turns out to be somewhat tricky because contiguous real pages may not be contiguous in physical memory. Thus, the VMM may need to convert an I/O request that spans multiple pages into multiple I/O requests, each referring to a contiguous block of physical memory. It may also be the case that I/O is to a real address that the VMM has swapped out of physical memory into its own swap space. In this case, the VMM must read it back into physical memory before the I/O operation can proceed.

The simultaneous operation of several virtual machines, along with the operation of the VMM itself, can degrade the performance of the system. In practice, therefore, additional considerations apply in order to enhance the performance of a program running on a virtual machine. An example of this is a policy in the VM/370 CP, which, when ready to activate a new VM, gives priority to the virtual machine likely to have most of its active pages already mapped to physical pages. Another example, used when the same OS is shared by several guest machines, is to employ a common mapping of the OS readonly pages. As VM/370 became popular, hardware features were added that allowed the performance of virtual machines to be comparable to that of native hardware. These will be described in more detail in **Section 8.5**.

8.3.3 Virtualizing an Architected TLB

When an ISA provides a software-managed TLB, then it is the TLB that must be virtualized by the VMM. To virtualize the TLB, the VMM maintains a copy of each guest's TLB contents and also manages the real TLB. Any instructions that modify the architected TLB are sensitive and are intercepted by the VMM so that it can keep the virtual copies up-to-date.

virtual copies up to date.

A simple way to manage the real TLB is for the VMM to rewrite the TLB whenever a guest VM is activated. That is, the VMM copies the VM's virtual TLB entries into the physical TLB, after appropriately translating the real addresses in the virtual TLB to physical addresses in the physical TLB. This is essentially the same as the shadow mapping in architected page tables. A problem with this scheme is that all the TLB entries must be rewritten each time control transfers from one guest VM to another (or to the VMM). This TLB rewrite incurs a fairly high overhead, especially for large TLBs.

An alternative method is to leverage the address space identifiers (ASIDs) that are normally included in each entry of an architected, software-managed TLB. These are intended to allow multiple processes to have address space mappings in the TLB simultaneously. There is an architected ASID register, and whenever the TLB is accessed as part of address translation, the current ASID register value must match the ASID value in a TLB entry in order for the TLB entry to be considered valid for the active process.

The ASID mechanism can be virtualized to allow the VMM to manage the TLB in a globally efficient manner. With this approach, each guest VM has a virtual ASID register, and the VMM maintains the real ASID register. At any given time, the VMM maps certain guest ASID values to real ASID values for entries that are present in the real TLB. Hence, some combination of guest address spaces from different guest VMs may be simultaneously present in the real TLB, under control and management of the VMM. If the VMM decides to put an entry in the TLB and a real ASID has not been assigned to the address space used by the entry, then the VMM must assign a real ASID to the new address space. This may involve deassigning the ASID from some other virtual address space and invalidating any TLB entries using the ASID. If a guest writes to its virtual ASID register, the write is a sensitive instruction that is intercepted by the

VMM. The VMM can then modify the virtual copy in memory and also change the real ASID with a real ASID value that maps to the virtual ASID.

Figure 8.16 illustrates this method applied to the example given earlier in **Figure 8.14**. The virtual TLBs are shown at the left. These correspond to the virtual-to-real page mappings for each of the guest VMs. The two programs both have entries in VMl's virtual TLB, distinguished by two different ASIDs. The VMM maintains the same real-to-physical map tables as are shown in **Figure 8.14**. The real TLB is shown on the right in **Figure 8.16**. There is one real TLB, and translations from the two VMs are simultaneously present. The ASID map table, maintained by the VMM, maps the virtual ASIDs to real ASIDs present in the real TLB. Note that program 1 in VM1 and program 2 in VM2 have the same virtual ASID (3) in their respective virtual machines. However, the VMM maps these ASIDs to two different ASIDs (4 and 9) so that they can be distinguished in the real TLB.

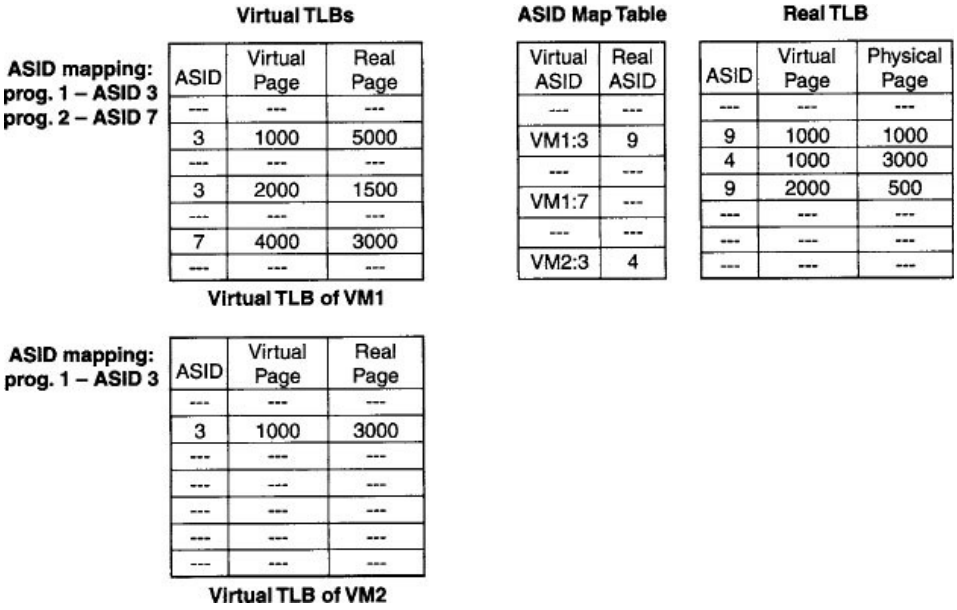


Figure 8.16 Virtualizing TLBs in a System That Implements Architected TLBs. *The system uses the same page mappings as given in* **Figure 8.14**. *The VMM maintains the virtual TLBs of the guest VMs (shown at left) and manages the real TLB by remapping virtual to real ASIDs. The ASID map table keeps track of the correspondence between virtual and real ASIDs.*

8.4 Resource Virtualization—Input/

Output

Virtualization of the input/output subsystem is one of the more difficult parts of implementing a system virtual machine environment. Each I/O device type has its own characteristics and needs to be controlled in its own special way. The difficulty of virtualizing I/O devices is compounded by the fact that the number of devices of a given type is often very large and continues to grow. Moreover, the number of I/O device types is also increasing. Virtualization of such a large number of devices is a challenge for any new VMM.

On the other hand, techniques to share I/O devices have been developed since the early days of time sharing. The proliferation of I/O devices is also a problem for conventional operating systems, which have developed abstractions to support a wide variety of devices and device types. It is possible to adapt many of these abstraction techniques for use in virtual machine systems.

The virtualization strategy for a given I/O device type consists of constructing a virtual version of the device and then virtualizing the I/O activity directed at the device. A virtual device given to a guest VM is typically (but not necessarily) supported by a similar, underlying physical device. When a guest VM makes a request to use the virtual device, the request is intercepted by the VMM. The VMM converts the request to the equivalent request for the underlying physical device and it is carried out. The next two subsections discuss techniques for virtualizing devices and carrying out virtual I/O activity.

8.4.1 Virtualizing Devices

A number of techniques can be used for virtualizing I/O devices, and the specific technique for virtualizing a given device depends on whether it is shared and, if so, the ways in which it can be shared.

Following are the common categories of devices.

Dedicated Devices

Some I/O devices, by their very nature, must be dedicated to a particular guest VM or at least are switched from one guest to another on a very long time scale. Examples of dedicated I/O devices are the display, keyboard, mouse, and speakers of a virtual machine user, as shown in **Figure 8.2**. In this case, the device itself does not necessarily have to be virtualized. Requests to and from the device could theoretically bypass the VMM and go directly to the guest operating system. However, in practice, the requests are usually routed through the VMM because the guest OS is running in the nonprivileged user mode. Interrupts from the device are first handled by the VMM, which determines that it came from a device dedicated to a particular guest VM and queues up the interrupt to be handled by the VM when it is activated by the VMM.

Partitioned Devices

For some devices, such as the disk, it is convenient to partition the available resources among the virtual machines. A very large disk, for example, can be partitioned into several smaller *virtual disks* that are then made available to the virtual machine as dedicated devices. A virtual disk is treated by a virtual machine exactly as a physical disk, except with a smaller capacity.

To emulate an I/O request for the virtual device such as a disk, the VMM must translate the parameters (e.g., the track and sector locations) into corresponding parameters for the underlying physical device(s), using a map maintained for this purpose, and then reissue the request to the physical device (e.g., the disk controller). Similarly, status information coming back from the physical device is used to update status information in the map and is transformed into appropriate parameters for the virtual device before being de-

livered to the guest VM.

Shared Devices

Some devices, such as a network adapter, can be shared among a number of guest VMs at a fine time granularity. Each guest may have its own virtual state related to usage of the device, e.g., a virtual network address. This state information is maintained by the VMM for each guest VM. A request by a guest VM to use the device is translated by the VMM to a request for the physical device through a virtual device driver. For a network, the VMM's virtualization routine will translate a request from some virtual machine to a request on a network port using its own physical network address. Similarly, the incoming requests through various ports will be translated into requests for virtual network addresses associated with the different virtual machines.

Spooled Devices

A spooled device is shared, but at a much higher granularity than a device such as a network adapter. An example of a device that is often spooled is a printer. Conceptually, the printer is solely under the control of the program that is printing until the completion of the printing task. This must be the case even if the program is swapped in and out several times during the printing process. *Spooling* allows this to be accomplished (**Madnick and Donovan 1974**). In the case of a printer, spooling involves the buffering of the lines to be printed from each program in an area of memory dedicated to that program. A buffer is closed only when the program that owns it sends a special signal to the printer. The operating system schedules the printing of the spool buffers as they close, ensuring that the output of each program is printed completely before the printer is allocated to the next program.

Virtualization of spooled devices can be performed by using a two-

level spool table approach, as depicted in **Figure 8.17**. The first level is within the operating system, with one table for each process active on the system, while the second level is in the virtual machine monitor, with one table for each guest virtual machine. An operating system's request to print a spool buffer is intercepted by the VMM, which copies the buffer contents into one of its own spool buffers. This allows the VMM to schedule requests from different virtual machines on the same printer.

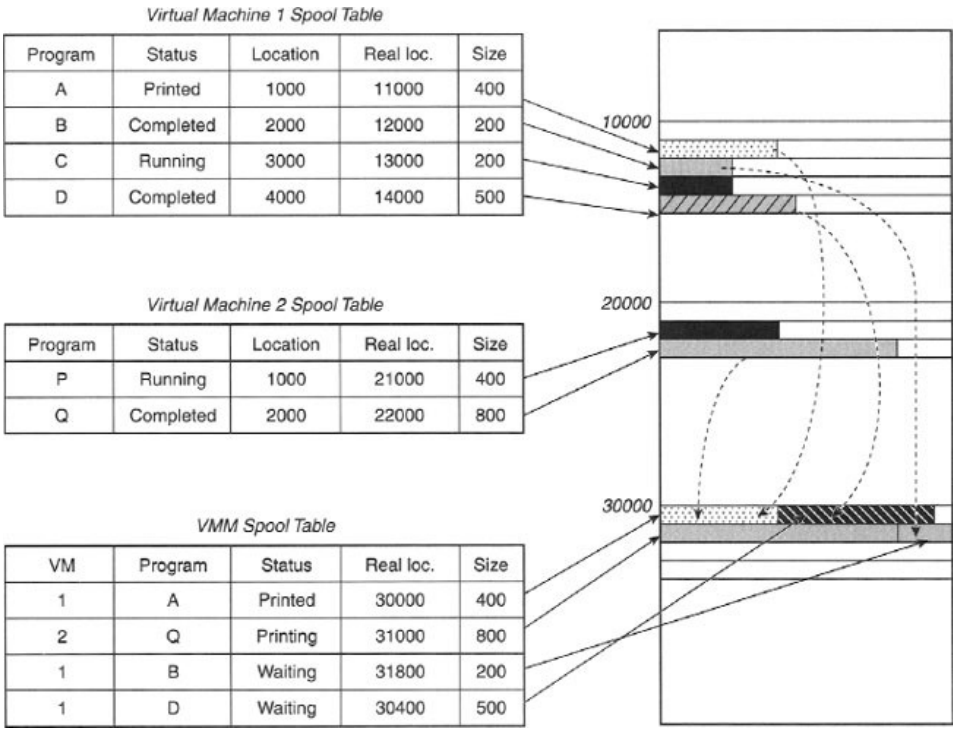


Figure 8.17 Spooling in a Virtual Machine System. *As programs are "completed," their spool areas are copied over to the VMM spool area. The corresponding entry in the VMM spool table is set to "waiting." At the end of printing, this status is changed in both tables to "printed." The spool area for the printed jobs can be recovered and reused.*

Various techniques could be employed to optimize the movement of large spool data between the virtual machine and the VMM. For example, the VMM could simply record the location of the spool data and its size, delaying the completion signal to the guest operating system until after the buffer has actually been printed. Another possibility is to have a single physical pool of buffers owned by the VMM, with each virtual machine having a virtual set of spool buffers allocated by the VMM. The practicality of this latter scheme de-

pends on the ability of the VMM to detect such allocation of buffers, and on the ability of the guest operating system(s) to invoke services provided by the VMM.

The virtualization of printers of this kind was more appropriate to the older line printers, which were generally expensive and attached closely to the main computer. Today it is more common to find multiple printers accessible over a network. These printers already have the software to buffer and spool jobs from various nodes in the network. Virtualization of the printer is simple in such an environment as long as the virtual machine is connected to the network and has it own distinct network ID. The printer software in this case is unable to distinguish between requests from real machines and requests from virtual machines.

Nonexistent Physical Devices

Virtualization makes it possible to provide support for virtual devices "attached" to a virtual machine for which there is no corresponding physical device. For example, each virtual machine could be associated with a network adapter that is used for communicating with other virtual machines on the same platform, without requiring an actual network adapter to be available on the platform. In such a situation the virtual machine monitor can simply intercept the I/O requests coming from the device drivers and emulate the transmission of the network packet, buffering the packet at the device driver of the appropriate destination virtual machine and causing an interrupt to be handled by that virtual machine.

8.4.2 Virtualizing I/O Activity

The overall I/O process is illustrated in **Figure 8.18**. As discussed in Appendix Section A.4.3, an operating system abstracts most of the peculiarities of hardware devices and makes these I/O devices accessible through the system call interface and the device driver inter

sible through the system call interface and the device driver interface. Then at a level even lower is the actual operation-level interface, where individual instructions interact with the I/O system, typically by placing device-specific addresses and data on an I/O bus. An application program makes device-independent I/O requests such as open() and read() through the system call interface. The operating system converts the device-independent requests into calls to device driver routines. A device driver operates in system mode and takes care of device-specific aspects of performing an I/O transaction. For example, when a file system uses a block device interface to write to a disk, the device driver converts a device-independent request to operations specific to the type of disk controller chip physically attached to the system.

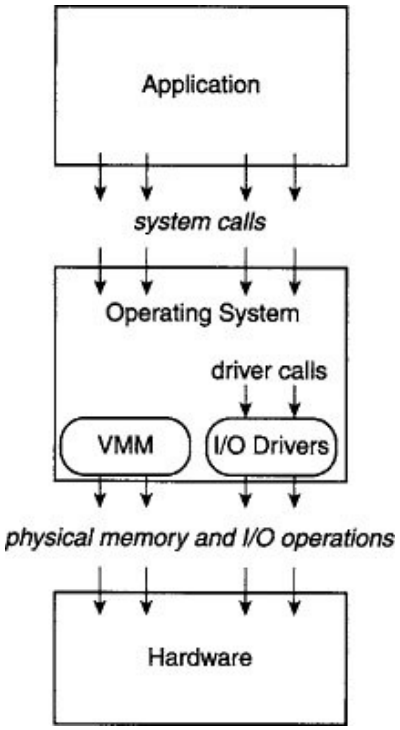


Figure 8.18 Major Interfaces in Performing an Input/Output Action.

When an I/O action is to be performed, part of the action is carried out by guest software and part by the VMM. In general, the VMM can intercept a guest's I/O action and convert it from a virtual device action to a real device action at any of the three interfaces: the *system call interface*, the *device driver interface*, or the *operation-level*

terminal interface, the device driver interface, or the operation level interface. We will discuss all three possibilities, in reverse order, beginning with the operation-level interface. Note that it is the VMM that finally interacts directly with the hardware device. Hence it is the VMM that either invokes the device driver or controls access to the device driver.

Virtualizing at the I/O Operation Level

Appendix Section A.3.5 contains a discussion of various types of I/O devices differentiated by the way the processor communicates with each device. With memory-mapped I/O, common in many RISC processors, the effect of sending a command to an I/O controller occurs by writing to or reading from special locations in the memory address space. These memory locations are protected by the operating system by making them inaccessible in user mode. On the other hand, processors such as the IBM System/360 and its successors as well as the Intel IA-32 provide special I/O instructions to signal the device controllers. User programs do not utilize these instructions directly; they invoke system routines that use privileged I/O instructions to complete the request.

The privileged (or protected) nature of the I/O operations makes them easy for the VMM to intercept because they trap in user mode. However, once intercepted, it may be difficult for the VMM to determine exactly what I/O action is being requested. A complete I/O action (e.g., a disk read) usually involves the device driver's issuing several I/O instructions or memory-mapped loads and stores of small granularity. These individual I/O operations work in concert—for the VMM to determine the higher-level I/O action to be performed, it must be able to "reverse engineer" the individual operations and deduce the complete I/O action. This could be extremely difficult in practice.

Virtualizing at the Device Driver Level

In a typical I/O request, as shown in **Figure 8.18**, a system call such as readO is converted by the OS into another call (or calls) to the device driver interface. If the VMM can intercept the call to the virtual device driver, it can convert the virtual device information to the corresponding physical device and redirect the call to a driver program for the physical device. This scheme is straightforward and allows virtualization at a natural point, but it requires that the VMM developer have some knowledge of the guest operating system and its internal device driver interfaces. This would be difficult if the VMM is called on to virtualize an arbitrary OS, with no knowledge of its internals.

In many practical situations, however, the VMM developer may be targeting specific guest operating systems. Today, for example, these might be Windows and Linux. In this case, special virtual device drivers (one per device type) can be developed for each of the guest operating systems. These drivers are part of the overall VMM package distributed to users. Then when a guest OS is installed, the virtualized drivers are installed at the same time.

This approach, when applied to a native VM system (see **Figure 8.5**), also requires that the VMM have real device drivers for each of the devices attached to the system. This problem can be simplified by "borrowing" the drivers (and the driver interface) from an existing operating system. For example, one could use the Linux drivers. In a hosted VM environment, the drivers in the host OS are used; such an approach is described in the next section.

Virtualizing at the System Call Level

In theory, the visualization process could be made more efficient by intercepting the initial I/O request at the OS interface, the ABI. Then the entire I/O action could be done by the VMM. To accomplish this, however, the VMM would need ABI routines that shadow the ABI routines available to the user. These routines would be different for

routines available to the user. These routines would be different for each type of guest OS. Writing the ABI emulation routines would have to be part of the VMM development task and would require much broader knowledge of the guest OS internals than writing drivers (as in the previous subsection). Furthermore, all interactions between the ABI emulation routines and other parts of the guest OS would have to be faithfully emulated. In general, this rather daunting task could be done only if the guest OS is well structured and understood intimately by the VMM developer.

Example: Network Virtuallzation

Figure 8.19 presents an example of the overall visualization process on a system having a host network adapter, with the virtual machines themselves implementing virtual network adapters similar to what was described in **Sugerman, Venkitachalam, and Lim (2001)**. Two cases are shown. In the first case, we assume that the virtual network interface card (NIC) is of the same type as the physical NIC in the host system. A request to send a message to an external computer system typically consists of a number of I/O operations, here assumed to be OUT or OUTS instructions in the IA-32 ISA. Associated with an OUT or OUTS instruction is a port number, say, 0xf0, which is a number in the range of IDs designated for the NIC in the guest system. Also associated with each port is a state bit that indicates whether the system should trap in response to an I/O request for that port.

As part of the state of the virtual machine, the VMM saves the permission map for all ports of the guest machine. The OUTS instruction is a privileged instruction, so when an OUTS instruction is executed, the system traps to the VMM, which examines the permission bitmap of the current guest. If the permission is set, the VMM converts the request to a new OUTS instruction for port 0x280, corresponding to the NIC port on the real machine. It also converts the virtual address pointing to the string of data to be moved to a virtual

address in the VMM space pointing to the same string of data or a copy. The VMM ensures that port 0x280 is enabled in its own permission map and issues the new OUTS instruction. When executed, now in the VMM's privileged mode, this OUTS instruction traps to the device driver installed on the VMM for the NIC, which then performs the desired transfer.

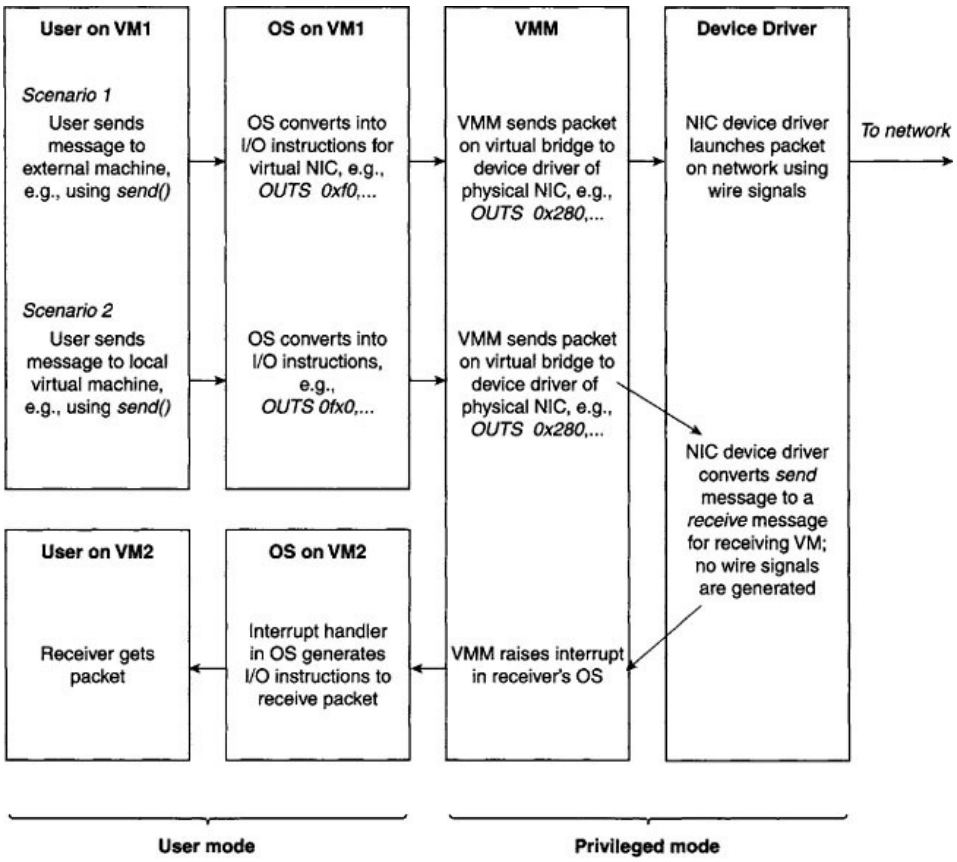


Figure 8.19 Virtualization of Network Interface Adapter.

The second case is where the VMM takes the original request and converts it into a different form, either because the physical NIC available on the system is different from the real NIC or because there is possibly a more efficient means of satisfying the request. When the physical NIC is not of the same type as the virtual NIC, the VMM implements a device driver for the virtual NIC that translates I/O operations directed to it (e.g., OUTS instructions) into other I/O operations directed toward the physical NIC. It is not always possible to perform such translations on the original sequence of I/O op-

erations one at a time. In these cases, it is necessary for the VMM to incorporate a device model of the virtual NIC, which first gathers enough information from a sequence of incoming I/O operations to generate the correct sequence of outgoing I/O instructions for the physical NIC.

In some cases, it is easier to modify the device driver for the physical NIC in order to handle the I/O operations directly. The second half of **Figure 8.19** shows a case where the desired communication is between two virtual machines on the same platform. In this case, the host device driver, invoked due to the trap of a translated OUTS instruction as described earlier, determines that the destination address of the message is another virtual machine running on the same platform and does not transmit the message on the physical wires connected to the NIC. Rather, it internally reroutes the message as a received message destined for the receiver's virtual machine and schedules an I/O interrupt for the destination guest.

It would also have been possible to do the rerouting in the kernel of the VMM without involving the device driver; such a solution is more efficient when implementable. This is the case with the minidisk cache (MDC) facility implemented on the z/VM, the current incarnation of the VM/370. The minidisk cache attempts to reduce disk I/O by caching data in processor storage. In many cases, especially those where the data is referenced by several virtual machines, the decrease in disk I/O more than makes up for the possible increase in the paging I/O. The VMM invokes a routine in the MDC layer that determines whether a disk request can be satisfied from the cache. If so, then the call to the physical device driver is completely avoided.

8.4.3 Input/Output Virtualization and Hosted Virtual Machines

As described in **Section 8.1.4**, a hosted virtual machine system is

one that works in conjunction with a host operating system running natively on the underlying hardware. In a dual-mode hosted virtual machine system, part of the virtual machine monitor runs natively on the hardware; another part of the VMM operates as a user application on the host operating system in order to invoke resource allocation services provided by the host. An I/O request from a guest virtual machine is converted by the native-mode portion of the VMM into a user application request made to the host. For example, if the driver on a guest virtual machine attempts to fetch sectors from a disk, eventually the user-mode portion of the VMM will issue a read() to the host operating system for the corresponding data. The host operating system satisfies the read () request by performing its own I/O, but the guest machine and the VMM remain oblivious to the physical details of satisfying the request. Similarly, if an I/O interrupt is received by the VMM, the interrupt gets thrown over to the host operating system and its device drivers before being received by a guest operating system.

An important advantage of a hosted virtual machine is that it is not necessary to provide device drivers in the VMM; the device drivers already present on the host operating system are used indirectly by the virtual machine system. This is particularly convenient in situations where the large number and variety of devices supported make it difficult to provide all device drivers for the VMM. The PC desktop environment is the prime example of such a situation.

In addition to the host OS, there are three additional components that form a dual-mode hosted virtual machine system.

- **VMM-n (native):** This component runs natively on the hardware and has characteristics similar to the VMM on a native virtual machine system. It is the component that intercepts traps due to privileged instructions or patched critical instructions encountered in a virtual machine. It may provide device drivers for a small set of devices that are either performance critical, or that do not have drivers already available in the host operating system.
- **VMM-u (user):** This component runs as a user-mode process on the host

operating system. As mentioned earlier, this component makes resource requests to the host OS, in particular, memory and I/O requests, on behalf of the native mode VMM. VMM-u makes these requests using system library functions supplied with the host operating system.

- **VMM-d (driver):** This component provides a means for communication between the other two components. This is done by making the VMM-n appear to the VMM-u as a device attached to the host operating system. The VMM-d is essentially a special device driver installed on the host operating system, and it provides the link from VMM-u to VMM-n. The only user program on the host system allowed to access the VMM-n "device" is the VMM-u component.

As stated earlier, the biggest advantages of a hosted VM are the convenience of installing a virtual machine system on a platform already running a commercially available operating system and the fact that the actual device drivers do not have to be incorporated as part of the VMM.

The dual-mode hosted virtual machine system has some disadvantages also. First, the VMM-n operates in privileged mode alongside the host OS; consequently, the VMM-n and host OS are capable of accidentally (or maliciously) affecting each other. This vulnerability is worsened because the host OS is typically not developed and debugged with the VMM-n in the system. Second, the allocation of resources is completely under the control of the host operating system. The effects of allocation policies in the VMM are less predictable because the VMM has insufficient knowledge and control over the use of resources.

There is also a performance disadvantage compared to a native virtual machine system because of the need to go back and forth between two "worlds," the VMM-n world and the host OS world. When such a "world switch" occurs, sufficient state of one world must be saved before handing control over to the other world. The performance degradation is more significant in an I/O-intensive workload than in a user-mode CPU-intensive workload.

8.4.4 Input/Output Virtualization in VM/370

The I/O organization on the System/370 uses an IOP-based model, as shown in Appendix Figure A.5d. The IOP gets commands from the processor, initiates I/O requests on behalf of the processor, controls the transfer of data directly between the I/O device and memory, and informs the processor by means of an interrupt when the I/O is completed. The System/370 IOPs are called *channels,* and communication between the OS and the channels occurs through a program built from special instructions called *channel command words* (CCWs). Each CCW includes an address in memory, the length of data to be transferred, and a command to an I/O device. The process of making a request to the channel involves building in memory a sequence of CCWs and then issuing a privileged start I/O (SIO) command that includes a pointer to the CCW. As far as the processor is concerned, execution of the SIO is completed once the device is detected and the I/O operation is actually started. On completion of the I/O operation, the issuing processor is served with an I/O interrupt.

In a virtual machine, an I/O request to a device takes a similar path, provided the device is physically implemented. The virtual machine sets up the CCW in its virtual address space. The SIO instruction is privileged and hence trapped. The CP (VMM) refers to the shadow page table to map the virtual address of the CCW to a physical address. Based on this CCW it builds a new CCW, translating the device address specified by the virtual machine to the address of the physical device to which it is mapped. The CP also indicates the translated physical storage location for the data to be interchanged with the I/O device and ensures that these pages are physically in memory. The use of paging complicates this translation process, because I/O commands that span more than a page may need to be broken up into multiple CCWs. The CP then sends an SIO command targeting the real I/O device. Since the CP works in privileged mode, this SIO gets transmitted to the channel and does not generate a trap.

Only when the CP returns from its SIO instruction does it unblock

the SIO instruction generated by the virtual machine. If there are delays in accepting the command due to, say, an I/O device's being busy, the CP could still schedule other virtual machines to run on the processor. When the I/O operation completes, the IOP generates an I/O interrupt. The CP intercepts this interrupt, reports the relevant information in the status table of the processor that generated the request, and reflects the interrupt to the operating system on the appropriate virtual machine, which then handles the interrupt in the usual manner.

A different scheme is adopted for slower I/O devices, such as printers and card readers. In these cases, the CP emulates these devices and generates software interrupts to the virtual machine while buffering the information on a disk. This spooling technique (discussed in **Section 8.4.1**) allows more efficient utilization of the resources when several virtual machines attempt to access them simultaneously.

Finally, VM/370 provides for I/O devices for which there are no physical counterparts. The representative device of this type was the *minidisk,* which allowed for virtual disks with far fewer than the 203 cylinders that disks provided in those days. This enabled an economical pooling of what was then an expensive resource among multiple virtual machine users.

8.5 Performance Enhancement of System Virtual Machines

Virtual machines can improve the utilization of hardware by sharing resources among multiple users, each provided with the illusion of owning all the resources in the machine. Unfortunately, this also raises the expectations of users, who now want performance on their workload similar to that provided by a complete machine.

Performance measurements on early virtual machine systems indicated that even in the absence of other users on the machine, it was difficult to get a performance that was better than 21% of its performance on native hardware on some benchmarks (**MacKinnon 1979**). In the following sections we examine the sources of performance degradation in virtual machines and examine hardware assists aimed at closing the performance gap.

8.5.1 Reasons for Performance Degradation

Performance on a virtual machine differs from that on the native hardware for several reasons.

- **Setup:** Before a virtual machine can be activated, there is overhead involved in initializing the state of the machine. Such initialization includes setting up the timing facilities, the registers, and the program counter.
- **Emulation:** As discussed in previous sections, not all guest instructions can be natively executed. Some guest instructions need to be emulated (usually via interpretation) by the VMM. Often such emulation is not restricted to just the sensitive instruction; it is sometimes convenient to emulate instructions around a sensitive instruction to reduce transitions between executing in emulation mode and in native mode.
- **Interrupt handling:** Interrupts generated by a program running on a virtual machine first have to be handled by the VMM, even though they may eventually be handled by the guest operating system.
- **State saving:** There is overhead involved in saving the state of a virtual machine when control needs to be transferred to the VMM.
- **Bookkeeping:** Often, the VMM has to perform special operations to reflect behavior equivalent to that of a real machine. For example, the accounting of time charged to a user would be different in a virtual machine compared with a real machine, where user-mode activity typically is charged to the user and system-mode activity is charged to the system.
- **Time elongation:** Some instructions require more processing than they would have needed in the native mode. For example, instead of accessing just the page tables, it may be necessary in some situations to access both shadow tables and the local page tables. This has the effect of making the average time taken for some memory references considerably longer than on a real machine.

Hardware extensions may be used to reduce one or more of these

effects. *VM assist* is the name given to a piece of hardware that improves the performance of an application when running in the virtual machine mode. One effect of a VM assist is to reduce the number of situations where a user application must enter system mode for some action to be taken by the VMM. Thus not all virtual machine environments benefit from VM assists. In particular, operating systems designed specifically for virtual machine execution, such as the CMS in the VM/370 systems, do not invoke privileged instructions as often as other operating systems. As another example, simple operating systems such as DOS do not access virtual address translation facilities and hence may not benefit from special hardware designed to improve the translation performance.

In the following subsections we discuss various hardware techniques that help improve the performance of virtual machines. **Section 8.5.2** discusses techniques to improve instruction emulation performance, while **Section 8.5.3** discusses hardware to specifically help the performance of other aspects of the virtual machine monitor. **Section 8.5.4** then looks at ways of improving the performance of the system when running in user mode as a guest. **Section 8.5.5** examines some specialized techniques to enhance the performance of specific types of guest systems. Finally, in **Section 8.5.6** we describe an interesting approach implemented in System/370 that uses hardware essentially to take over a number of VMM functions.

8.5.2 Instruction Emulation Assists

This class of assists improves the performance of virtual machine applications in the areas that are the cause of fundamental overheads, namely, those situations that need emulation under VMM control. Recall that in most such situations, the execution of a privileged instruction causes an interrupt that is first handled by the VMM. The VMM emulates the instruction using a routine whose operation depends on whether the virtual machine is supposed to be executing in system mode or in user mode.

executing in system mode or in user mode.

An example on the System/370 is the assist for the load PSW (LPSW) instruction. As mentioned in **Section 8.2.1**, this instruction traps in user mode in order to prevent the manipulation of privileged resources such as the problem state bit by the user. With the assist, the hardware (via microcode) performs the entire action of checking the state of the virtual machine, determining which action to take and performing either the full action if the virtual machine is in system mode or the restricted action if in user mode. The physical resources in the machine are modified by a hardware-assisted instruction if they hold the corresponding virtual machine resource, as in the case of the general-purpose registers. Otherwise, the corresponding location in the state table held in memory is modified. These actions are identical to those that would have been performed by emulation of the instruction in the VMM. Hence the hardware assist depends on knowledge of the implementation of the virtual machine system. In fact, the VM assist implementation provides an additional bit, bit 1 of control register 6, to supply information on whether the guest virtual machine is operating in privileged mode or user mode.

The LPSW still remains a privileged instruction, and because the virtual machine is running in user mode, the overhead due to the trap itself is not eliminated. However, the LPSW hardware assist not only reduces the time taken in emulating the instruction, but also eliminates the overhead due to switching context from the virtual machine to the VMM and back again. Hardware assists of this kind improve the total performance of the system by reducing the number of instructions actually emulated by the VMM.

8.5.3 Virtual Machine Monitor Assists

Another way of improving the performance of a virtual machine environment is to use assists for improving performance of the VMM.

Here are some of the assists of this nature that have been used on the System/370.

- **Context switch:** Using hardware to save and restore registers and other machine states when switching context from a virtual machine to the VMM and back.
- **Decoding of privileged instructions:** Recall that all privileged instructions trap when running in the virtual machine mode, whereas they trap only in the user mode in a native environment. Since privileged instructions are seldom used in user-mode code, the trapping of these instructions on a native machine is not a significant source of overhead. On the other hand, in a virtual machine mode, even occasional privileged instructions encountered during the execution of the guest operating system could cause a significant overhead in performance. Hardware assists, such as decoding these instructions, normally performed in VMM software, go a long way toward alleviating this overhead. Note that unlike the instruction emulation assists mentioned earlier, this assist only helps with certain critical parts of the emulation, with the rest of the emulation still being carried out in software.
- **Virtual interval timer:** Most operating systems depend on the interval timer to schedule jobs. A guest operating system cannot emulate this function in exactly the same way as in a native mode. The best that can be done is to decrement the virtual counter by some amount estimated by the VMM from the amount that the real timer decrements. More accurate timer functionality needs hardware support. The VM/370 system requires that the virtual timer for each virtual machine be located in location 80 of page 0. Hardware ensures that while the guest VM is running, this location is decremented every time the real timer is decremented. In addition, the hardware assist presents a timer interrupt to the virtual machine when the virtual interval timer turns negative.
- **Adding to the instruction set:** In order to improve the performance of the VMM, the System/370 also introduced a number of new instructions that are not a part of the ISA of the machine. The nature of operations performed by these new instructions is specific to VM/370, because their inclusion was guided by an analysis of its commonly executed parts. Here are some examples of these instructions.
- Obtain free space from free storage area
- Return space to free storage
- Page lock
- Page unlock
- Translate virtual address and test for shared page
- Invalidate segment/page table

The VMM can detect the presence of these instructions in an imple-

mentation by examining the contents of control register 6. Various bits of this register denote which assists are available on the machine. If some specific assist instruction is not present, the machine ignores the instruction and executes the normal software routine it would have executed in the absence of such assists.

8.5.4 Improving Performance of the Guest System

The classical concept of *virtualization* postulates that a guest system in the virtual machine environment should not be aware of the fact that it is working in such an environment. It is the job of the VMM to ensure that privileged instructions and interrupts are handled in the same way as in a native environment. This is a requirement if nothing can be changed in software running on a real machine in order to make it operate on a virtual machine.

On the other hand, it is possible to get an improvement in performance if a guest OS knows whether it is currently executing natively or in a virtual machine environment. When there is a choice, an OS executing fewer privileged instructions will observe a smaller degradation in performance when executed on a virtual machine. Alternatively, an OS aware of the presence of a VMM could enjoy a performance benefit by relegating some functions to the VMM or by providing the VMM with additional information to carry out its actions. Such an exchange of information is referred to as *handshaking* (**MacKinnon 1979**). Handshaking is a programming technique that works in conjunction with hardware assists. A key feature of the System/370 that makes handshaking effective is the ability of the guest operating system to send a message to the VMM. The principal technique to achieve this is by the use of the DIAGNOSE instruction. Different forms of the instruction are defined to send different types of messages between the operating system and the VMM. Variants of the DIAGNOSE instruction are generally nonarchitected—they are

implementation-dependent features rather than extensions to the ISA.

Many of the improvements that have resulted from handshaking involve eliminating the duplication of function between the guest operating systems and the VMM. We list a few examples here.

- **Nonpaged mode:** The guest operating system disables dynamic address translation and defines its real address space to be as large as the largest virtual address space it needs. Thus, page frames are mapped to fixed real pages. Translation still occurs from these real pages to the physical pages on the native system. Hence the system as a whole still runs with dynamic address translation turned on. However, improvement in performance results from not having to deal with the bookkeeping of two levels of translation. Further, the guest operating system no longer has to exercise demand paging. The working set of the system as a whole is managed by the VMM, which continues to perform demand paging. Thus, not only is double paging eliminated by this scheme, the effects of potential conflicts in paging decisions by the guest operating system and the VMM are eliminated.
- **Pseudo-page-fault handling:** As discussed earlier in this chapter, a page fault in a virtual machine system can occur not only because of a page fault in some virtual machine but also because of the replacement of a page by the VMM to accommodate a page of a different virtual machine. In a conventional virtual machine system, on the occurrence of the latter type of page fault, the VMM would make the current virtual machine inactive and activate another virtual machine. Control is given back to the original virtual machine when the page fault has been handled. In *pseudo-page-fault handling,* when the page fault is caused by a missing entry in the VMM page table, the VMM attempts to improve fairness by giving control back to the same virtual machine, along with a special page-fault indicator, allowing the virtual machine to schedule an alternative process itself. This is useful in situations where the guest operating system is equipped to handle multiprogramming effectively and has enough threads that it can use to hide the latency of the page fault. The operating system can disable the feature when the level of multiprogramming is low. Pseudo-page-fault handling is enabled by a special Set Pagex CP command.
- **Spool files:** In the absence of any special mechanisms, the guest operating system in the virtual machine closes a spool file and puts it in its spool buffer. It issues I/O commands to dispatch each file in the buffer in turn to the I/O device, e.g., a printer. In order for this to work when multiple virtual machines are using the same I/O device, it is necessary for the VMM to intercept the I/O commands

the same I/O device, it is necessary for the VMM to intercept the I/O commands, spool these files in its own buffer along with requests from other virtual machines, and schedule them for printing. Rather than intercept the I/O commands and decipher that the virtual machine is attempting to send a job to the printer, handshaking allows the virtual machine to signal the VMM that a file is ready. In response, the VMM picks up the spool file (or, better still, a pointer to the file) and continues as before to merge this file into its own buffer.

- **Inter-virtual-machine communication:** The objective here is to send data between two virtual machines. Typically, communication between two physical machines involves the processing of message packets through several layers at the sender's side before the message gets physically sent on wires to the receiving machine, once again after being processed through several layers. This process can be streamlined, simplified, and made considerably faster if the two machines are virtual machines on the same host platform. On the System/370, this was achieved by supporting a special virtual machine running a special operating system called the Remote Spooling Communications System (RSCS). The RSCS uses the spool buffer belonging to the VMM either to manage data that may need to be transferred to another virtual machine on the same platform or to stage the data before sending it through network adapters to some remote machine. The RSCS later gave way to more efficient ways of performing inter-virtual-machine communication. In one case a virtual device called a channel-to-channel adapter (CTCA) was used to move data from one virtual machine to another using inexpensive move operations, rather than I/O operations. In another, called Virtual Machine Communications Facility (VMCF), it was possible for data to be communicated from one virtual machine to several virtual machines simultaneously simply by the use of storage-to-storage transfers. The interesting aspect of many of these solutions is that they have the potential not only of reducing the overhead due to virtual machine operation but also of actually improving communication performance between distinct machines.

The notion that the performance of a virtual machine system can be improved by making modifications to the guest operating system has received renewed interest through what is being called *paravirtualization* (**Whitaker, Shaw, and Gribble 2002**). Paravirtualization presents a virtual machine interface to a system that is similar but not identical to the underlying native hardware. As embodied in the Xen system (**Barham et al. 2003**), the paravirtualization interface is specifically designed to work around features that make it difficult to virtualize the underlying ISA. Xen specifically targets the IA-32 ISA, which, as we have seen earlier, has critical instructions that

make it difficult to design an efficient virtual machine. The Xen system takes an existing operating system, such as Linux or Windows XP, and makes modifications to the machine-dependent parts of the system to eliminate the need to perform complex virtualization tasks such as code detection and patching or the maintenance of shadow page tables. As an example, Xen takes advantage of the fact that popular operating systems today use only rings 0 and 3 of the IA-32 privilege levels by modifying the guest operating system to run in ring 1. Privileged instructions present in the operating system code are paravirtualized by requiring them to be validated and executed within the virtual machine monitor, which runs in ring 0. Xen aims to restrict changes to just the operating system—an application binary works unchanged in Xen. The Xen researchers report that the number of lines of the Linux IA-32 code base that needed to be changed in order to paravirtualize it was less than 3000, or 1.36% of the total. The result is a virtual machine system that performs at a level of 90+% across all benchmarks, compared to a native Linux implementation.

8.5.5 Specialized Systems

Further optimization of a virtual machine system is possible if the VMM has some knowledge about, or access to, the internals of the guest operating system. This is the case when the guest operating system is not a proprietary closed box. Following are some examples of this type of performance enhancing hardware (**Gum 1983**).

- **Virtual-equals-real (V = R) virtual machine:** In this mode, the host address space representing the guest real memory is mapped one-to-one to the host real memory address space. One advantage of such a mapping is the improvement of performance of channel programs accessing large, data-spanning multiple pages. Normally, contiguous real pages of the virtual machine need not be mapped to contiguous locations in physical memory. Hence such a channel program must be retranslated into multiple channel programs, each accessing a contiguous region of the physical memory. The V = R mapping eliminates the need to perform such retranslation, thus reducing the overhead experienced by channel programs

retranslation, thus reducing the overhead experienced by channel programs.

- **Shadow-table bypass assist:** As described in **Section 8.3**, the shadow page tables for a guest differ from the actual guest page tables only in that the real addresses in the former point to host physical addresses rather than real addresses in the guest machine. The maintenance and update of these tables constitute a fair amount of overhead in a virtual memory system. These shadow page tables can be dispensed with—the guest page tables can point directly to physical addresses if the dynamic address translation hardware is allowed to manipulate the guest page tables. On System/370, however, it is necessary to treat accesses to page 0 in a special way because this page is mapped distinctly for each virtual machine. The shadow-table bypass assist takes this into account in hardware.
- **Preferred-machine assist:** The idea of this assist is to allow a guest operating system to operate in system mode rather than user mode. This eliminates the special treatment required to execute most of the privileged instructions in the guest operating system. Special hardware is needed to support this capability because even though the VMM operates in the same mode as the guest operating system, it has to be protected from unintentional or malicious tampering by a guest operating system. Certain events may still need to be examined to determine whether they are actually meant for the virtual machine currently executing, for the VMM, or for some other guest virtual machine. For example, external I/O interrupts may not be destined for the current guest virtual machine and may need to be handled by the VMM.
- **Segment sharing:** When multiple virtual machines loaded with the same guest operating system are running on a system, it is possible to improve performance by sharing the code segments of the operating system among the virtual machines, provided the operating system code is written in a reentrant manner. In particular, if the code segments contain only read-only information, sharing a single copy can alleviate TLB pressure on the system. While most programs today satisfy the reentrant property, older operating systems often did not—they modified code segments, usually for storing small pieces of data but sometimes to modify code behavior. Sharing in such cases would require the VMM to perform expensive checks to ensure that a virtual machine does not use code modified by another virtual machine. The System/370 provided hardware to perform such segment protection, eliminating the need for software checks.

Hardware assists that enhance the performance of specific guest systems inherently take advantage of knowledge of system details. Often on the System/370, a particular hardware assist may work only for a specific guest operating system; in fact there may be multiple versions of the same assist, depending on characteristics of the virtual machine. Some specialized hardware assists can leave infor-

virtual machine. Some specialized hardware assists can leave information in guest OS-accessible data structures revealing to the guest OS that it is running in a virtualized mode rather than natively. This, in itself, is usually not a cause for concern, but it does compromise the original principle that any software running on a virtual machine should not know, and hence should not exploit the fact, that it is not running natively.

8.5.6 Generalized Support for Virtual Machines

With the success of the virtual machine philosophy, IBM looked into ways to formally include features in the ISA that improved the performance of virtual machine systems in general. This led to the idea of an *interpretive execution facility (IEF*), which provides a way for the processor to directly execute most of the functions of the virtual machine in hardware. Thus, in a sense, interpretive execution is an extreme case of a VM assist, where almost all instructions are assisted. Besides further improving the performance of the virtual machine, interpretive execution benefits virtual machines running all types of guest operating systems and also makes the behavior of virtual machine programs more predictable across a range of architectures and implementations.

An IEF can be viewed as a transparent integration of the various techniques used to assist the performance of virtual machines. The function of the VMM is distributed more cleanly over hardware and software. In fact, the traditional software part of the VMM gives up control to the hardware IEF part through an instruction called *start interpretive execution* (SIE). Before it does so, it takes care of any pending interrupts, determines the virtual machine to be dispatched next, sets a pointer to a table containing the architected state of the virtual machine, and loads some of the resources (such as the general-purpose registers) with corresponding values from the table.