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.

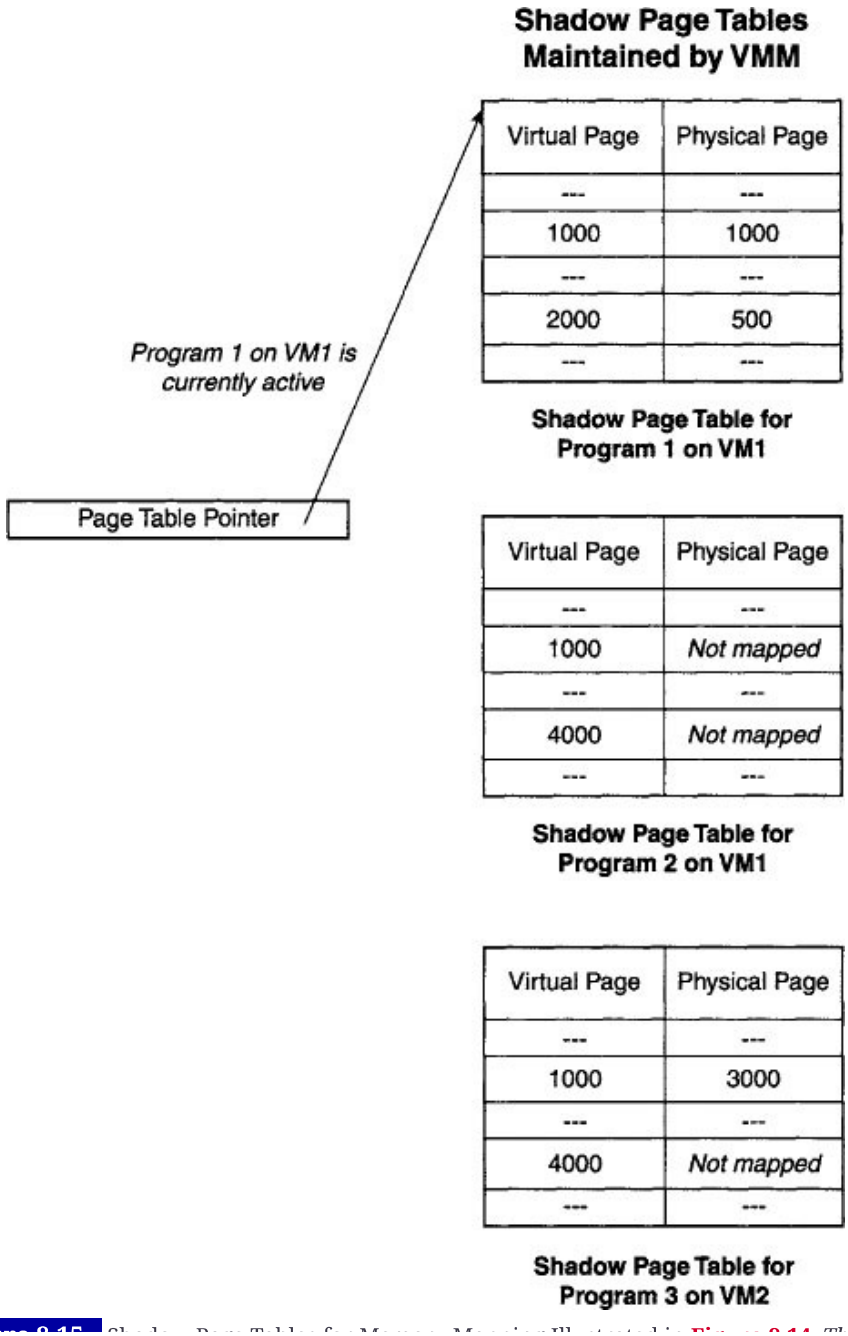


**Figure 8.15** Shadow Page Tables for Memory Mapping Illustrated in **Figure 8.14**. *The shadow tables are used by hardware in performing address translation. The VMM manages the shadow tables and is responsible for setting the page table pointer register.*

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.

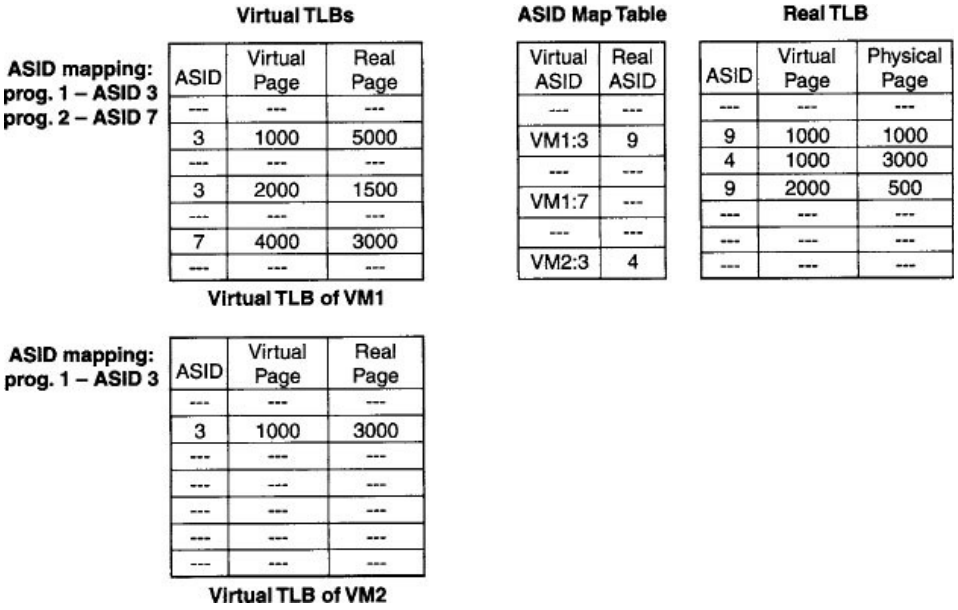



**Figure 8.16** Virtualizing TLBs in a System That Implements Architected TLBs. *The system uses the same page mappings as given in* **Figure 8.14**. *The VMM maintains the virtual TLBs of the guest VMs (shown at left) and manages the real TLB by remapping virtual to real ASIDs. The ASID map table keeps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

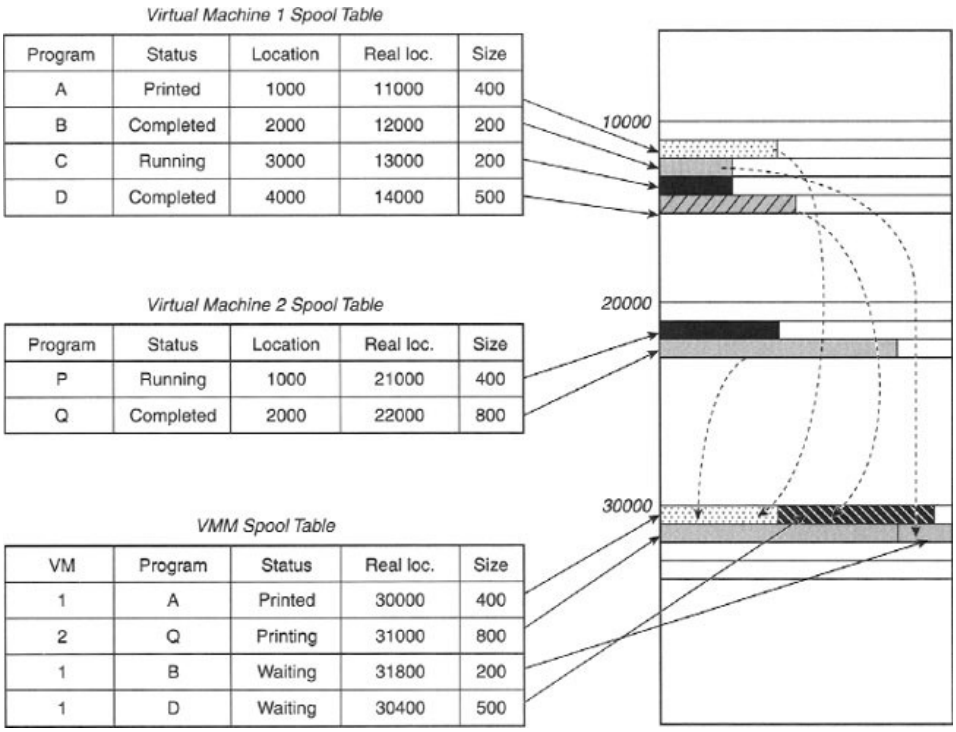



**Figure 8.17** Spooling in a Virtual Machine System. *As programs are "completed," their spool areas are copied over to the VMM spool area. The corresponding entry in the VMM spool table is set to "waiting." At the end of printing, this status is changed in both tables to "printed." The spool area for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

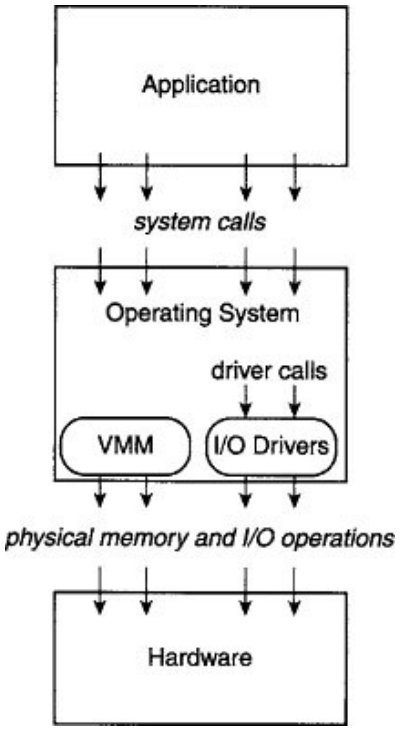



**Figure 8.18** Major Interfaces in Performing an Input/Output Action.

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*

*system call 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.

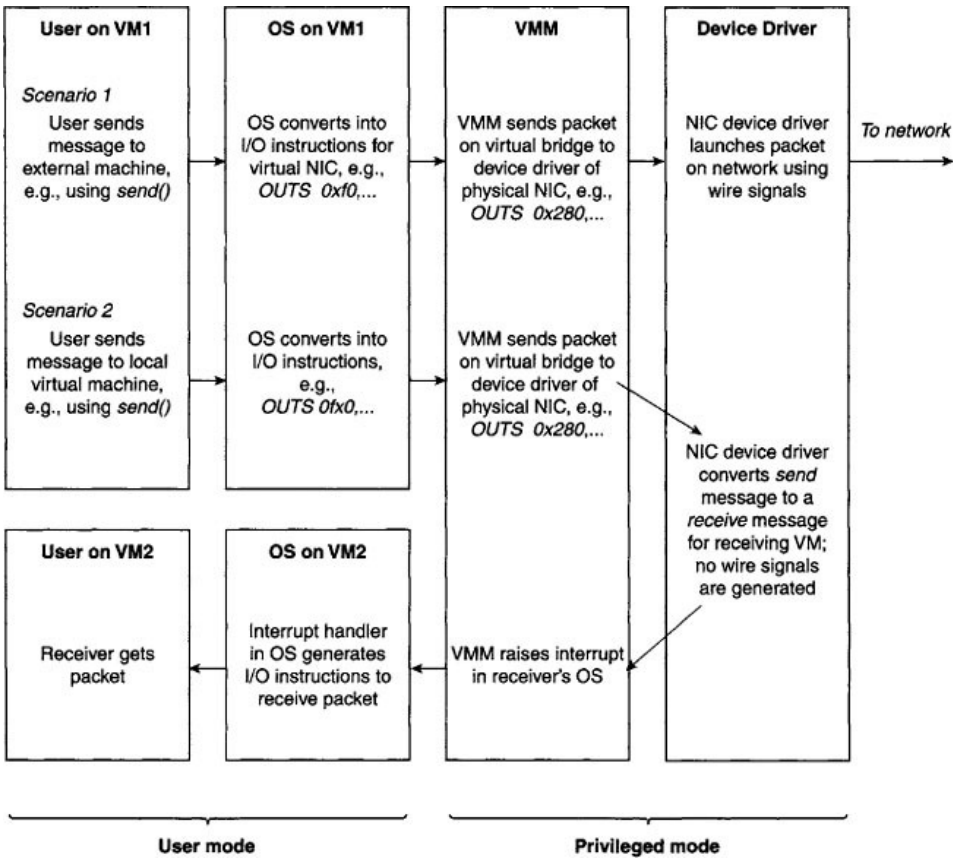



**Figure 8.19** Virtualization of Network Interface Adapter.

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.

In the IEF mode, most privileged instructions are executed directly in hardware. The notable exceptions are I/O instructions and some of the more complex, though infrequent, instructions. Once the IEF gets control, it starts executing the instruction stream from the virtual machine, directly executing many of the instructions and interpreting others in hardware. As shown in **Figure 8.20** there are two ways in which the hardware interpretation of instructions may be suspended and control returned to the software part.

1. An interrupt may be generated that is determined by the IEF hardware to be meant for either the VMM or another virtual machine. A return from such an interrupt points to the SIE instruction that was executed by the VMM.

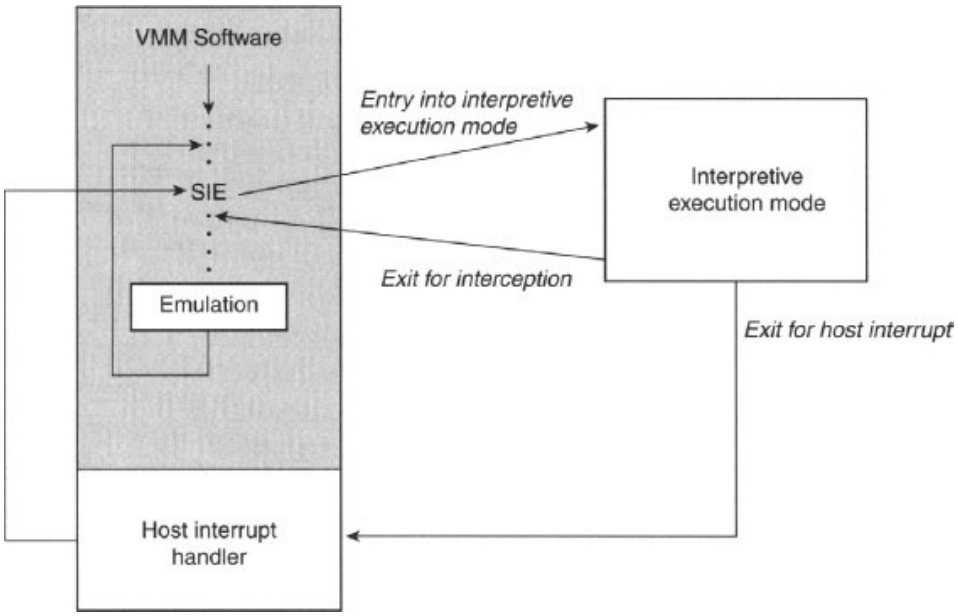



**Figure 8.20** Interpretive Execution Entry and Exit. *Execution of an SIE instruction causes the processor to enter the interpretive execution mode. Exit from this mode can occur either because an interrupt needs to be handled by the host or due to one of several conditions that require servicing by the VMM. In the former case, the machine returns back to the interpretive execution mode. In the latter it proceeds to the next instruction after the SIE.*

2. Control can pass to the instruction following the SIE instruction. This is referred to as *interception.* Interception occurs because of the following reasons.

a. An instruction is encountered whose interpretation is not supported in hardware.
b. An exception occurs during the execution of an interpreted instruction.
c. An instruction is encountered in a mode that forces it to be handled in software, as in a traditional virtual machine system.

d. An externally set interception condition is detected.
e. Some special case is detected; e.g., a guest operating system has entered the wait state.

Before control returns to the software part of the VMM, the IEF hardware must update the table containing the state description of the virtual machine so that it can resume operation at a later time. It must also store some information about the reason for giving control back to software, including parameters that would be needed for software emulation.

**TABLE 8.1** Improvement in Performance of Virtual Machines Resulting from the Introduction of Hardware VM Assists on the IBM System/370 (MacKinnon 1979)

| | | Model 135 | | Model 145 | | Model 158 | |
|---|---|---|---|---|---|---|---|
| | | DOS/VS | VS1 | DOS/VS | VS1 | VS1 | VS2 |
| Elapsed time (seconds) | Native | 2788 | 3035 | 2150 | 1418 | 1386 | 572 |
| | Virtual Machine | 8172 | 11598 | 4520 | 4089 | 3769 | 2696 |
| | Virtual Machine with VM Assist | 4226 | 4063 | 2723 | 2024 | 2004 | 1149 |
| Relative Batch Throughput | without VM Assist | 0.34 | 0.26 | 0.48 | 0.35 | 0.37 | 0.21 |
| | with VM Assist | 0.66 | 0.75 | 0.79 | 0.7 | 0.69 | 0.5 |
| Reduction in supervisor state time through VM Assist | | 74% | 89% | 73% | 86% | 82% | 69% |
| Reduction in elapsed time through VM Assist | | 48% | 65% | 40% | 51% | 47% | 57% |
| Reduction in total number of instructions simulated by VM/370 through VM Assist | | 87% | 95% | 86% | 94% | 91% | 74% |

For good performance, the IEF relies on many of the hardware assist features described earlier in this section. By having special hardware assists for the timer and for the time-of-day clock and by using bypass techniques to reduce the overhead of memory address translation, the IEF comes closer to its goal of making the behavior of a program on a virtual machine very similar, only slightly slower, compared to its performance on a native platform. This is evident from **Table 8.1** (MacKinnon 1979), which shows the improvement in performance through the use of assists for three different implementations of the System/370 architecture and for three different guest operating systems: DOS/VS, OS/VS1, and OS/VS2 SVS. It can be observed that the difference in performance of a program running natively versus in a virtual machine decreases significantly through the addition of special hardware features that support virtualization.

The interpretive execution facility has evolved considerably since its original introduction. In essence, more and more function was added in hardware to eliminate most of the overhead involved in supporting virtual machines. A comprehensive discussion of the various new features can be found in **Osisek, Jackson, and Gum (1991)**.

## 8.6 Case Study: VMware Virtual Platform

A popular virtual machine infrastructure for IA-32-based PCs and servers is the VMware Virtual Platform (**VMware 2000**). As mentioned earlier, the VMware system is an example of a hosted virtual machine system. More recently, VMware has included a native virtualization architecture embodied in a product called the VMware ESX Server, which is claimed to provide better resource control, scalability, and performance, but at the expense of full support for all types of hardware. Our discussion here is limited to the hosted system, which has been renamed the VMware GSX Server (**VMware 2001**). We outline next the various reasons that motivated this development of a hosted system rather than a native system.

In comparison to a native VM infrastructure for a large mainframe, such as the VM/370, the development of a VM infrastructure for a commodity IA-32 environment faces several challenges.

- The IA-32 grew up as an inexpensive microprocessor, never intended to be used in large systems supporting multiple users. Among other things this led to an ISA that includes a number of features difficult to virtualize efficiently. On the other hand, the IBM VM systems have existed almost from the beginning of the System/370 line. The IBM architecture team had close interactions with the team developing the VMM (because they worked at the same company), and therefore they avoided features that could hamper the development of virtual machines on its platforms.
- The other difference between the current commodity IA-32 environment and the

IBM System/370 environment is the openness of the system architecture, which has led to a proliferation of different configurations of PCs. Thus, while the VM/370 developers had only to understand the system configurations that IBM shipped to its customers, IA-32 VMM developers must anticipate any of the myriad configurations that a multitude of PC system vendors may ship. The situation is made even more daunting by the proliferation of different I/O devices that are available for users to add to their systems.

- VMware, as a separate company from either a hardware developer, such as Intel, or an operating system company, such as Microsoft, faced an additional challenge. It had to ensure that its VMM software can be easily installed and used. In particular it cannot expect its users to wipe out an existing operating system to install VMware software and then reinstall the old operating system over the VMM. This, in fact, directly influenced the architecture of the VMM developed by VMware.

The VMware Virtual Platform presents a view to the user depicted in **Figure 8.21**. At this level it looks similar to the VM view presented to an IBM System/370 user. However, as opposed to the native nature of the virtual machine system of VM/370, the Virtual Platform is a hosted system. It depends on the existence of a host operating system, such as Windows or Linux, running on the bare hardware to perform certain critical functions. As we have seen in **Section 8.4.3**, a principal advantage of a hosted system is that the VMM can avail itself of any I/O device the host operating system supports with relatively little work.

As discussed in **Section 8.4.3**, a dual-mode hosted virtual machine system must have three components for its VMM: (a) the VMM-n, which is the component working in a privileged native mode, (b) the VMM-u, the component that runs as an application on the host operating system, and (c) the VMM-d, the component that enables communication between the VMM-n and VMM-u. On the VMware system, these three components are respectively named VMMonitor, VMApp, and VMDriver. At any given time, the processor executes either in the host operating system environment or in the VMMonitor environment. Transfer of control between the two worlds is facilitated by the VMDriver and involves saving and restoring all user and system visible state information on the processor. As opposed to

and system visible state information on the processor. As opposed to the view presented to the user shown in **Figure 8.21**, the structural view of the VMware system is as shown in **Figure 8.22**.

Even though the Intel IA-32 architecture features four levels of protection, rings 0 through 3, most operating systems on the IA-32 simply lump the kernel and services, including the device drivers, into ring 0. Thus, for all practical purposes, it is adequate to consider the architecture as having a privileged level, largely consisting of code running in ring 0, and a user level, consisting of code running in ring 3.

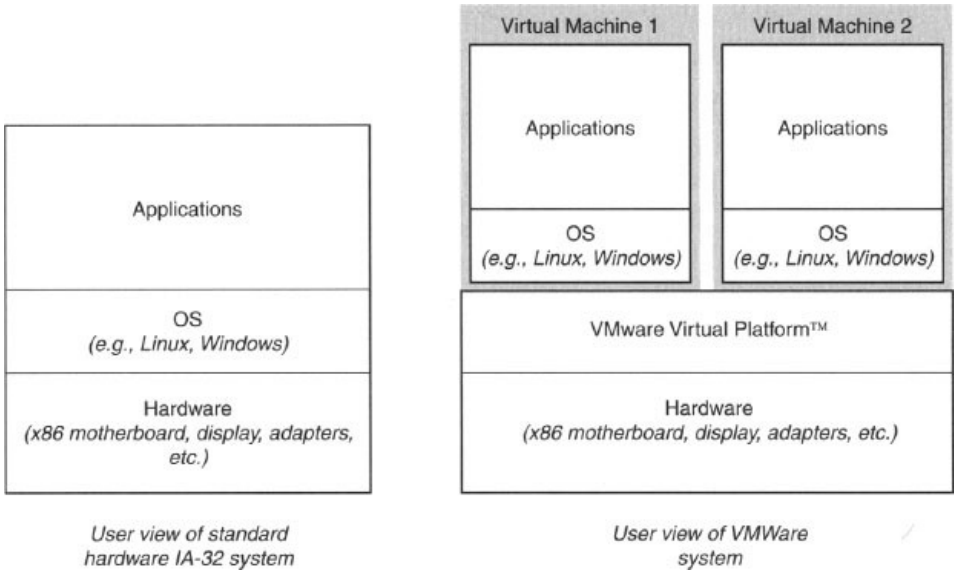



**Figure 8.21** User View of the VMware System.

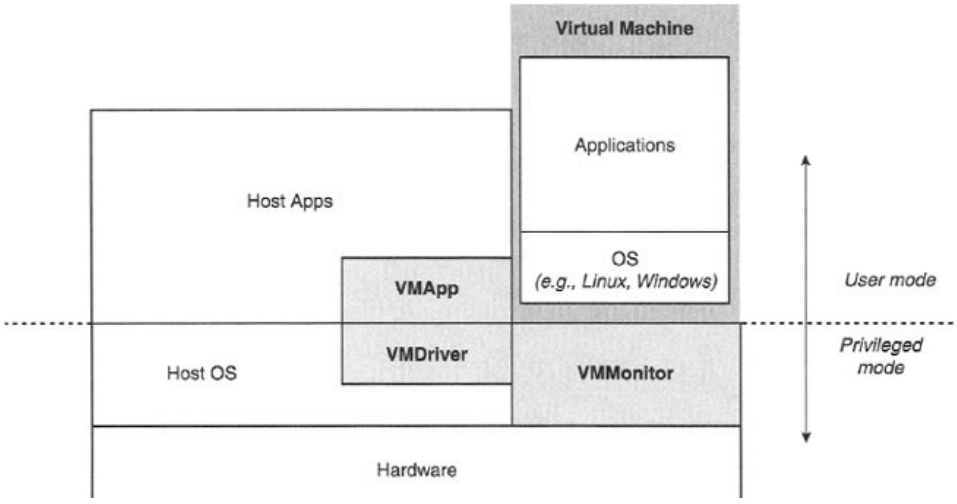




**Figure 8.22** Components of the VMware System. *VMMonitor runs in privileged mode. VMDriver is a device driver installed in the host operating system and hence also has system privileges. VMApp runs as a user-mode application on the host operating system.*

VMMonitor operates in privileged mode, but VMApp, which is simply an application that requests services from the host operating system, operates in the user mode. In a sense, VMMonitor can change its personality from being a privileged controller of resources to being just another application on the host operating system. It does this efficiently through VMDriver, a special device driver installed on the host operating system and hence having the same privileges as the host system.

## 8.6.1 Processor Virtualization

The Intel IA-32 architecture is not efficiently virtualizable, i.e., a VMM cannot be constructed in the strict sense of Theorem 1 in **Section 8.2.1**. Researchers (**Robin and Irvine 2000**) have identified 17 IA-32 instructions that are critical, i.e., sensitive but not privileged. Hence, a virtual machine system for the IA-32 architecture must be a hybrid virtual machine system. VMware presumably includes mechanisms for discovering these critical instructions and for patching (or caching) them, as described in **Sections 8.2.3** and **8.2.4**.

The 17 critical instructions fall into two broad categories.

- **Protection system references:** These instructions reference the storage protection system, memory system, or address relocation system. The hazard here is the possibility of some virtual machine accessing locations not in its virtual memory. An example of this is the MOVE instruction, which moves a value from a general-purpose register to the CS register, the control register that specifies the current privilege ring number in bits 0 and 1. In order to offer some protection, an instruction such as mov ax, cs executed in the user mode disallows the CS register to be loaded but does not generate a trap. Instead of trapping, the instruction generates a no-op and hence is not efficiently virtualizable.
- **Sensitive register instructions:** These instructions attempt to read or change

resource-related registers and memory locations, such as a clock register or interrupt registers. An example is the POPF instruction mentioned in **Section 8.2.1**. This instruction pops a word from the top of a stack in memory, increments the stack pointer by 2, and stores the value in the lower 16 bits of the EFLAGS register. One of the bits in the EFLAGS register is IF, the interrupt-enable flag, which, in the user mode, is not modified when POPF is executed. To understand why this is a sensitive instruction, consider the case where a guest operating system executes a POPF instruction. The operating system may require that the IF bit be changed; but if it is running in the user mode under the virtual machine, the IF bit is not changed. The guest operating system could later take erroneous actions because the flag bit had not been set as expected. In a hybrid virtual machine implementation, such an instruction would be discovered by the VMM and patched to generate a trap. The trap handler in the VMM would then emulate the instruction and achieve the result expected by a guest operating system.

Let us now examine what happens when a guest operating system in a virtual machine attempts to use a critical instruction. The piece of IA-32 code shown in **Figure 8.23** is an example of real code (**Rosenblum 2000**). In this code, the flags are saved on stack using a pushfd instruction before a small piece of code is executed that could potentially change them. When they are restored, the expectation is that the flags will be exactly the same as before being pushed on the stack. In the privileged mode, the popfd would adequately restore the flags to their original state. However, when the code is executed in user mode, as it would be on a virtual machine, not all flags will be properly restored, unless the instruction is emulated.

```
pushfd
cli
mov         cax, (0x824)
cmp         cax, 1
jc          5
mov         (0x900), cdx
popfd
add         cdx, cax
```

**Figure 8.23** Sample Intel IA-32 Code Using the popfd Instruction.

In order to perform this emulation, VMMonitor scans the instruc-

tion stream being executed and detects the presence of the popfd instruction in the stream. The monitor then substitutes this instruction with another set of instructions that takes the processor into the privileged state and emulates the action of the original code. The emulation would include a check for the current mode of operation and execute one of two actions for popfd, depending on the mode. Specifically, if the virtual machine itself was supposed to be executing in the system mode, the emulation of the popfd instruction must include the loading of the EFLAGS register or, more accurately, the virtual EFLAGS register belonging to the executing virtual machine. On the other hand, if the virtual machine is not in system mode, the emulation must ensure that certain bits, such as the IF bit, are not modified in the virtual EFLAGS.

The setting of the IF should always be in control of the VMM, so that when a virtual machine is swapped out and another one brought in, the IF is not swapped. For each virtual machine there is a location in memory that corresponds to the EFLAGS register, and the emulation of activity on the EFLAGS register is carried out through instructions operating on the virtual flags in memory, rather than by loading the EFLAGS register and performing the action directly. Emulation of the IF in the EFLAGS register implies that any instruction that reads or modifies this bit must also be emulated. The cli instruction in the example of **Figure 8.23**, which clears the interrupt flag, therefore, is also emulated by VMMonitor.

## 8.6.2 Input/Output Virtualization

The difficulty of virtualizing I/O in popular IA-32-based platforms was a primary reason that VMware developers chose to implement the monitor in a dual-mode hosted VM style. The PC platform supports many more devices and many more different types of devices than any other platform. This has resulted in a massive number of device drivers written by all segments of the industry, including de-

vice hardware vendors, operating system developers, software developers, and even users. VMware takes a multifaceted approach to solving the I/O virtualization problem (**Sugerman, Venkitachalam, and Lim 2001**).

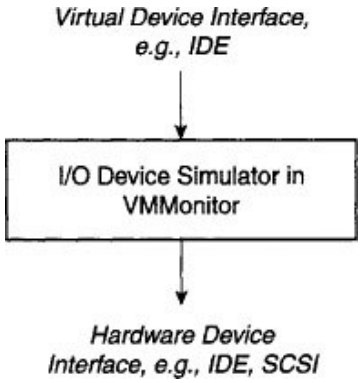



**Figure 8.24** Mapping a Virtual Device Interface to a Hardware Device Interface in the VMware Virtual Platform.

### Emulation In VMMonitor

The first way to virtualize an I/O device is to emulate the device in the VMMonitor. If the device to be virtualized already has a physical counterpart on the host, the job of emulating it is simply one of converting the parameters in some virtual device interface (VDI) into parameters of the actual hardware device interface (HDI). Thus, the i n and out I/O instructions are intercepted by the VMMonitor and converted to appropriate i n and out instructions for some actual physical device (**Figure 8.24**). A situation where such a scheme would be used is that of converting a command to read a block from an IDE drive to one that reads a block from a SCSI drive. Unfortunately, such conversion requires some knowledge of the ports and memory map locations associated not only with the SCSI drive, but also with the IDE drive supported in the guest operating system.

### Using the Services of the Host Operating System

The VMware developers discovered that while some interfaces, such

as the IDE interface, appear to be well standardized across the industry, there is a noticeable variation in devices following the SCSI, LAN, or graphics standards. This called for a different approach to emulating these interfaces. Instead of converting the in and out I/O instructions to new in and out instructions, they are converted into a set of library service requests for the host operating system, which invariably already supports the needed device drivers.

This technique is illustrated in **Figure 8.25**. The method relies on the dual personalities of the Virtual Platform. The VMMonitor contains device models for each of the virtual devices supported. When a guest I/O instruction is encountered, it traps to the VMMonitor code that alters the state of the device model corresponding to the requested device. If the requested device is not natively supported by VMMonitor but is supported by the host OS, the request is converted into a host OS call. For example, if the host operating system is Windows NT, it is converted into a *win32* system call. The VMMonitor, through VMApp, now acts as a well-behaved application under Windows NT. When the application returns from this system call, the control gets back to the VMMonitor and then into the application running on the virtual machine. VMMonitor uses this approach for virtualizing the floppy disk drive, the CD-ROM drive, the sound card, the serial port, and the parallel port. In measurements taken on a real system, the overhead of switching from one mode to the other was determined to be only tens of microseconds.

The dual personality assumed by the VMMonitor confers advantages beyond reducing the complexity of supporting a large variety of I/O devices.

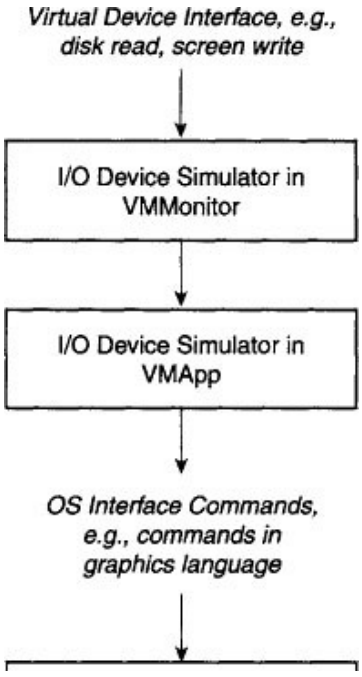
- As long as the operating system provides a reasonable suite of services to perform I/O, any operating system can be used as the host. All devices supported by that operating system would automatically be available to VMMonitor.
- The VMMonitor need not limit its access of the host operating system to its I/O features. All services available on the host operating system can be taken

features. All services available on the host operating system can be taken advantage of because VMMonitor installs on the host operating system just like any other application. In particular, it can use the file system services of the host, say, to emulate disks.

- When making a transition from an old operating system to a newer one, it may be beneficial to use the old operating system as the host, with the new one working in a virtual machine. Performance-critical applications can be run on the host directly during the transition period.

### New Capability for Devices Through Abstraction Layer

VMApp has the ability to insert a layer of abstraction above the physical device. This allows the incorporation of new functionality on the device that may not have been possible or convenient with the original device. For example, the disk on a virtual machine in VMware can be treated either as a raw disk or as a file on the host operating system. In the latter case, it then becomes possible to have "undoable" disks, where recent actions taken on the disk can be undone. This also makes it possible to save or discard entire sessions by providing a commit feature that performs explicit commits of disk writes at the end of the session. Such a capability is particularly useful, for example, for a test session or if one needs to discard a session because of an insecure access.



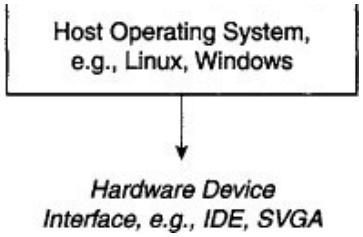



**Figure 8.25** Use of the Operating System Library Function Interface to Map a Virtual Device Interface to a Hardware Device Interface.

This abstraction layer can also provide a way to reduce performance losses due to virtualization. For example, a virtual Ethernet switch is provided in VMware between a virtual NIC (network interface controller) and a physical NIC to allow multiple virtual machines to share a physical NIC, similar to what was described in connection with **Figure 8.19**. Different IP addresses can be given to each of the virtual machines. The virtual machines sharing the physical NIC can communicate with each other through this virtual LAN, exactly in the manner of a more traditional LAN interconnection between physical machines. The latency of communication between these virtual machines can actually be made lower than that encountered in a physical system because the LAN is emulated using special communication paths that are local and more efficient than the standard Ethernet path.

A different type of abstraction is exemplified by the provision in VMware for an alternative user interface. In general, user interface controls such as the video card, the keyboard, and the mouse are physical resources that are virtualized in a manner similar to other physical resources, such as general-purpose registers. A user on a virtual machine takes direct control of these devices just as he/she would on a real machine. Thus the entire screen of the monitor can be under the control of the virtual machine. In addition, VMware also provides the capability to make the user-interface display appear as a window on the user interface of the host operating system, e.g., as one of the windows of a Windows desktop. This capability allows the user to access applications on both platforms in a convenient way. Using special communication paths between VMMonitor

client way. Using special communication paths between VMMonitor and the host operating system, it is even possible to cut and paste between a window on the host and an application on a virtual machine.

### 8.6.3 Memory Virtualization

VMMonitor virtualizes the physical memory of a virtual machine by using the host operating system to allocate or release the real machine's physical memory. The guest operating system does standard demand paging, though physical memory pages for the virtual machine are assigned by the host operating system. Paging requests are not directly intercepted by the VMM. They are converted into disk read/writes by the guest OS exactly as they would be on a real machine. These disk read/writes are translated by VMMonitor to requests on the host operating system through VMApp. The requests are actually made to appear by VMDriver as large DMA requests on the host operating system. The activity of applications on the host determines whether or not such a DMA request is satisfied from the memory of the host or results in paging on the host. The standard replacement policies of the host could end up replacing critical pages, thereby degrading performance of the virtual machine system in the presence of other host applications. In order to alleviate this problem, VMDriver pins some of the critical pages of the virtual memory system in physical memory, especially those pages belonging to the VMMonitor's active working set.

## 8.7 Case Study: The Intel VT-x (Vanderpool) Technology

In describing the VMware Virtual Platform in **Section 8.6**, we indicated several of the problems in virtualizing the Intel IA-32 architecture. We also saw how the VMware GSX server tackled the virtualizability problem to provide virtual machine capability on this popu-

lar platform using a hosted VM system. However, the various techniques used to finally achieve virtualization come at a cost—complexity of code and performance overhead. Some of the overhead, particularly that of scanning and patching the code to get around the presence of critical instructions in the instruction set, will also be present in native virtual machine implementations like the VMware ESX server. This was the motivation behind the current trend towards paravirtualization, where the interface presented by the virtual machine is not identical to that of the architecture of the underlying processor, but rather simplified to eliminate the effect of critical instructions as much as possible.

The performance overhead of handling the sensitive instructions in the VMM still remains, even with paravirtualization. In **Section 8.5** we described various ways to speed up virtualization through hardware enhancements of the processor. VM Assists and the Interpretive Execution (IE) capability went a long way in improving the performance of the z/VM system on the IBM zSeries mainframes. Intel recently developed a capability, the VT-x, or Vanderpool, technology for IA-32 processors, that is conceptually similar to the IBM's IE capability, and promises to enhance the performance of IA-32 virtual machine implementations.

The main feature of VT-x is the inclusion of the new VMX mode of operation. When in VMX mode, the processor can be in either VMX root operation or VMX non-root operation. In both cases, all four IA-32 privilege levels (rings) are available for use by software. VT-x, in effect, provides four new less privileged rings of protection for execution of guest software, in addition to the usual four rings for use by the VMM. The behavior of the processor in VMX root operation is largely similar to its function in a normal processor not incorporating the VT-x technology, the main difference being the inclusion of a set of new instructions called the VMX instructions. The behavior of the processor in non-root operation is limited in some re-

spects from its behavior on a normal processor in order to support virtualization. The limitations are such that critical shared resources are kept under the control of a monitor running in VMX root operation. This limitation of control of resources extends also to non-root operation in ring 0, which, in normal processors, is the most privileged level. Thus the intention is for the VMM of a virtual machine system to work in VMX root operation, while the virtual machine itself, including the guest operating system and applications, works in VMX non-root operation. Because VMX non-root operation includes all four IA-32 privilege levels (rings), guest software can run in the rings in which it was originally intended to be run (i.e., the guest operating system kernel can run in ring 0 and guest applications can run in ring 3).

## 8.7.1 Technology Overview

The typical sequence of operations in a virtual machine system is shown in **Figure 8.26**. A processor which has been turned on in a normal mode can be made to enter VMX root operation by issuing a vmxon instruction. The virtual machine monitor running in root operation sets up the environment for each virtual machine and initiates the virtual machine by issuing a special instruction, vmlaunch. Various situations, especially attempts to reconfigure critical resources shared with other virtual machines or with the VMM, cause the processor to relinquish control to the VMM in root operation. The processor may also exit from the virtual machine explicitly through a vmcall instruction. Re-entering a previously launched virtual machine is accomplished through the lower-cost vmresume instruction, which puts the processor back in non-root operation. Thus, in the steady state, the processor should be working largely in VMX non-root operation with occasional short excursions to the VMM. Finally, when the processor wishes to leave VMX operation, it will bring down each of the virtual machines in turn and return to the normal mode by issuing a vmxoff instruction.

This is a high-level view of how a virtual machine system can be created on a VT-x enabled IA-32 processor. There are many aspects which need to be taken care of in launching and exiting from a virtual machine; the interested reader is referred to Intel's Preliminary Specification document (**Intel 2005**) for more details on this topic. We limit our discussion below to the concepts underlying the technology.

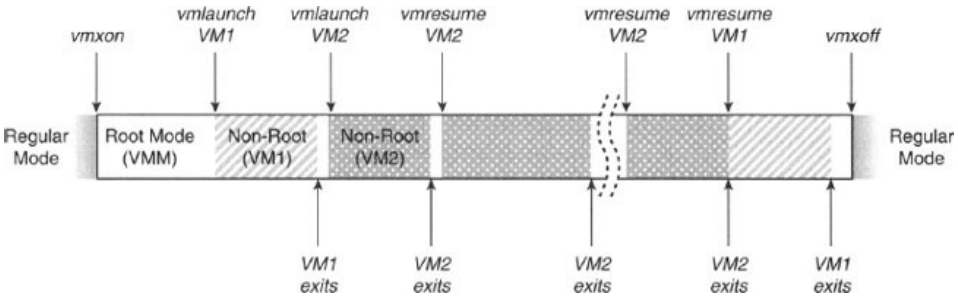



**Figure 8.26** Transition Between Operation Phases with the Intel VT-x Technology. *White areas represent the operation in root mode of the virtual machine monitor (VMM). The cross-hatched and dotted areas represent the regions in time when the two active virtual machines are executing.*

## 8.7.2 Capabilities of the Technology

A key aspect of the VT-x technology that allows faster virtual machine systems to be built is the elimination of the need to run all guest code in the user mode, essentially by providing a new mode of operation specifically for hypervisors and VMMs. For code regions that do not contain instructions that affect any critical shared resources, the hardware executes as efficiently as it would have on a normal machine. Since hardware keeps track of the virtual machines that are active, even larger regions can be executed at full speed by temporarily mapping the shared resources to the corresponding resources in the virtual machine. It is only in the few cases where this is not possible that a certain degree of emulation must be performed by the VMM. Thus, once in the virtual machine, the exits back to the monitor are far less frequent in the hardware case than in software virtualization.

The other major source of overhead in a software-based solution is the maintenance of state information. Various tricks in the software, especially mapping the frequently used registers to hardware register elements, reduce the overhead of maintaining and changing state. VT-x technology provides hardware that allows almost all of the state-holding data elements to be mapped directly to their native structures during virtual machine execution. Moreover, the technology provides a formal data structure called the VMCS (virtual machine control structure) that encapsulates all the information needed to capture the state of a virtual machine or to resume a virtual machine. The hardware implementation can take over the tasks of loading and unloading the state from their physical locations, without requiring the VMM to perform expensive load and store operations on large quantities of data.

The VT-x hardware is thus able to reduce the overhead in a classic native VM system considerably. It also eliminates the need for paravirtualization, thereby allowing more standard versions of operating systems to be used in virtual machine systems. The vmcall instruction, which can be used by a VMX non-root guest to communicate with the root mode VMM, can be used as a mechanism to pass hints and data that can help the VMM to perform more intelligent allocation of resources.

VT-x can be adapted also to construct a hosted VM system. Since VMX root operation has been designed to be as close as possible to the normal operation of a processor, it is possible to take a standard implementation of a host OS and make it run in VMX root operation. The VMM, also running in VMX root operation, is then in the same environment as the host OS, and has the same privileges as the host OS, just as in a typical hosted-VM implementation like VMware's.

An important reason for the choice of a hosted system for the VMware GSX server was the need to virtualize the large number of existing devices and device drivers that exist in the IA-32 PC world.

This need still remains; the VT-x technology does not address I/O virtualization. It must be noted also that the VMX root operation is not exactly identical to normal operation of the processor. There are restrictions imposed in VMX operation, such as the inability to reset the CRO.PE and CRO.PG bits to 0, which prevent a host OS in VMX operation from running in unpaged protected mode or in real-address mode. Modern applications will not suffer because of this limitation; however, on legacy systems these modes may have to be supported either through emulation in the VMM, or by executing in the normal mode after leaving VMX operation through a vmxoff instruction.

## 8.7.3 Maintenance of State information

We now turn our attention to the mechanics of state maintenance in the VT-x technology. As mentioned before, the state of a virtual machine is maintained in the VMCS data structure. This structure is fully specified, with the various fields precisely defined. The VMCS can be manipulated only by hardware, or by software running in root operation. The VMCS corresponding to the currently executing VM is identified by a pointer, the VMPTR, which contains the 4KB-aligned physical address of the VMCS. Multiple VMCS structures corresponding to different virtual machines can be active, though only one can be executing at a time on a given logical processor. (There may be more than one “logical” processor in a multithreaded system.) The VMPTR can be modified using the vmptrld instruction in root operation. Similarly, the contents of the VMCS can be accessed only in root operation, and only through vmread and vmwrite instructions. Regular memory load and store operations may not be used to access data in the VMCS because the format used to store the VMCS is not architected, and may change from one implementation to the next.

The different types of information stored in the VMCS are shown in

Table 8.2. The state information that is stored for the guest and the host goes beyond the conventional architected state and includes items like the architecturally hidden part of each segment register. This is needed to maintain a behavior of a virtual machine similar to its behavior on a normal machine. The various control fields determine the conditions under which control leaves the virtual machine (VM exit) and returns to the VMM, and define the actions that need to be performed during VM entry and VM exit. For example, since the MSRs on a processor are implementation-dependent, one of the control fields determines how many model-specific registers (MSRs) must be saved on VM exit, and another field determines the physical address where these registers must be saved.

**TABLE 8.2** Information Stored in the Virtual Machine Control Structure (VMCS)

| | | |
|---|---|---|
| **State Area** | Guest State | Register State |
| | | Interruptibility State |
| | Host State | Register State |
| **Control Area** | VM Execution Controls | Pin-based Execution Controls |
| | | Processor-based Execution Controls |
| | | Bitmap Fields |
| | | etc. |
| | VM Exit Controls | Control Bitmap |
| | | MSR Controls |
| | VM Entry Controls | Control Bitmap |
| | | MSR Controls |
| | | Controls for Event Injection |
| **VM Exit Information** | Basic Information | VM-Exit Information |
| | | Vectoring Event Information |
| | Other Exit Information | Due to Event Delivery |
| | | Due to Instruction Execution |

An important part of the VMCS area is the VM-exit information area. This area contains fields that inform the VMM the reason for an exit, as well as other information needed to service the event that caused the exit.

Instructions may cause exits, either unconditionally, e.g., instructions disallowed in non-root mode, or conditionally, e.g., a rdtsc instruction, which causes an exit only if a special control bit “RDTSC Exiting” has been set. There are several such control bits described

in the specification document (**Intel 2005**), that essentially allow the VMM to set up virtual machines with different kinds of behaviors.

## 8.7.4 An Example: The rdtsc Instruction

We conclude this section by demonstrating how the hardware supports a virtual machine executing the "Read Time-Stamp Counter" instruction, rdtsc. According to the IA-32 architecture, the time-stamp counter is maintained in an MSR called the IA32_TIME_STAMP_COUNTER. Execution of the rdtsc instruction causes the 64-bit value in this counter to be transferred to a specified pair of general-purpose registers. However, if the TSD (Time-Stamp Disable) bit of control register CR4 is set and the privilege level is not 0, a protection-mode exception will result and the counter will not be read.

A flow chart showing the various conditions that arise is shown in **Figure 8.27**. The first thing the hardware checks is the TSD bit in the physical CR4 register that is mapped to the CR4 of the currently active VM. If the TSD bit in this register is set and the privilege level is not level 0, the operation must cause a protection-mode exception in the guest. The hardware directly examines a bitmap, called the Exception Bitmap, in the Processor-Based VM-Execution Controls area of the VMCS for the currently executing VM. This bitmap is a 32-bit vector that contains one bit for each IA-32 exception. If the bit corresponding to the mentioned protection-mode exception is set, the hardware forces a VM exit to allow the VMM to take appropriate action. Otherwise, the exception is directly delivered to the guest as in a processor operating in the normal mode. Thus the hardware avoids the need for the VMM to intercept the exception, inspect it, and reflect it back to the guest OS, as typically done in software VM implementations.

