the fault. In doing so, it consults the various internal data structures that contain information about the cells and physical resources that each virtual machine was using.

The entire system needs to participate in order to ensure that all affected virtual machines are properly purged from the system and that all references to the affected virtual machines are eliminated from the data structures kept in each cell. This is a considerably more elaborate process than that needed on a physically partitioned system—and is the cost for improving the utilization of the system under normal operating conditions.

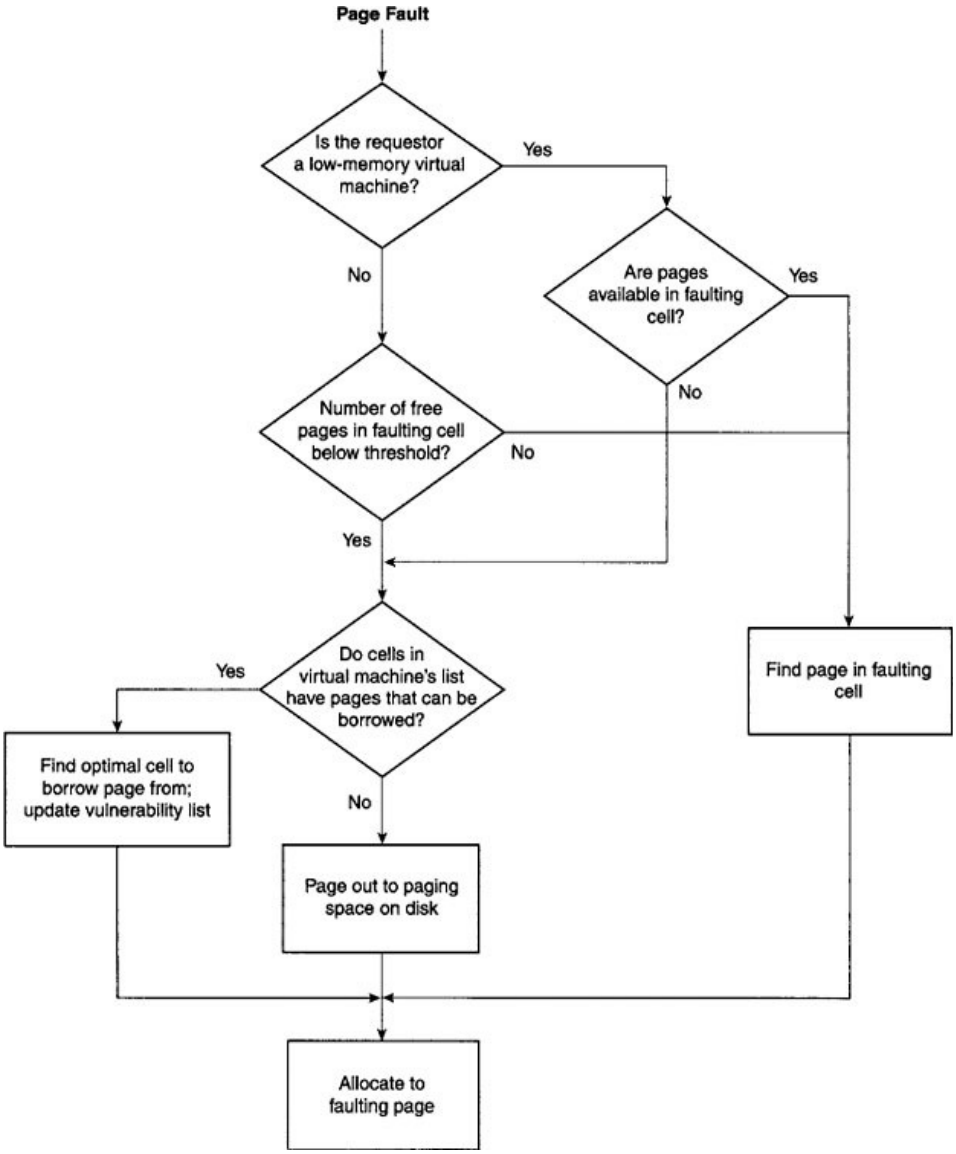


Figure 9.15 Cellular Disco's Actions on a Page Fault. *As far as possible, the system avoids using the paging disk. The requests of virtual machines with low memory requirements are*

satisfied locally, if possible. When choosing a cell to borrow from, a cell from the vulnerability list is chosen, if available.

The Cellular Disco research suggests that it is possible to implement fairly-sophisticated virtualization and fault-containment mechanisms in an efficient way. Its general philosophy of using physical boundaries as strong hints for the purposes of fault containment rather than as rigid walls provides a degree of flexibility, especially in situations where an application is willing to trade off some of the fault-containment characteristics for better performance. While the research was conducted in the context of system virtual machines, many of the ideas could eventually find application in hypervisor-based logically partitioned systems.

9.5 Virtualization with Different Host and Guest ISAs

All the schemes described thus far have involved situations where the ISA of the virtual system is identical to the ISA of the host system. When the two ISAs are different from each other, added complexities are introduced.

- The instructions of the target ISA must be dynamically emulated by the host system. The translation of instructions from the ISA of the virtual machine to the native ISA is an operation that must not be visible to the virtual machine or the programs running on it, including the operating system. The emulation techniques needed to achieve this are those described in connection with the implementation of process VMs (**Chapters 2**–**4**) for most ISA features and codesigned VMs (**Chapter 7**) with respect to some system ISA features, such as page faults.
- The memory model of the target system, particularly the coherence and memory ordering rules, must be observed on the virtual system. These involve additional actions on the part of the host system. The host could handle this either in hardware, in the firmware, or in emulation software contained in the trusted monitor layer, i.e., either in the virtual machine monitor or in the hypervisor.

As described earlier, there are two different ways in which multi-

processors are generally configured. The processors can be connected in a loosely coupled cluster configuration, where not much other than I/O is shared between the processors. Alternatively, the processors can be configured in a shared-memory configuration, with all processors in the multiprocessor configuration sharing the same memory. Implementing a cluster virtual machine on a native cluster system is fairly straightforward if each target virtual processor is mapped to one real processor. This case is shown in **Figure 9.16**. Since memory is not shared between nodes in such a cluster, it is sufficient to virtualize each processor using techniques given earlier in this book. Virtual node communication is through messages that can be translated to similar messages on the host.

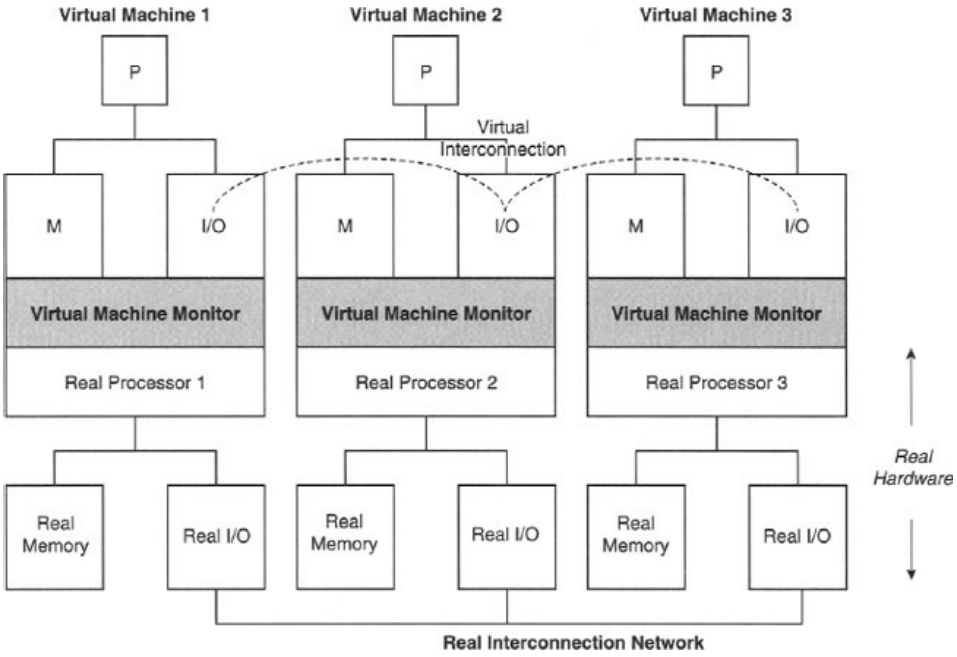


Figure 9.16 Virtual Uniprocessor Cluster on a Real Uniprocessor Cluster.

When the system to be virtualized is a shared-memory multiprocessor, the situation becomes more complex. Because the memory is shared between the processors, the monitor on each processor must handle shared-memory requests exactly the way they would be handled on a native machine. So, for example, if the guest virtual machine supports coherent shared memory, the monitors running on the separate processors must ensure that information written to any

the separate processors must ensure that information written to any shared-memory location is made visible to all processors that share that location. If the underlying native system is itself a shared-memory system, then the hardware mechanisms that already exist on the system to support coherence can be used to ensure coherence for the virtual system. This is the case, for example, in the FLEX-ES system from Fundamental Software Inc. (FLEX), which provides a virtual SMP for an IBM System/390 system as a user application under Linux on an Intel IA-32 based SMP. The basic IA-32 SMP is enhanced with special hardware adapters, as shown in **Figure 9.17**, to emulate the functions of the System/390 communication and channel I/O adapters. Since the underlying Intel IA-32 platform also implements memory coherence, the coherence expected by the virtualized mainframe system is trivially supported. While the memory ordering rules of the Intel IA-32 are not exactly identical to those of the System/390, the differences can be taken care of in the VMM layer without significant degradation of performance.

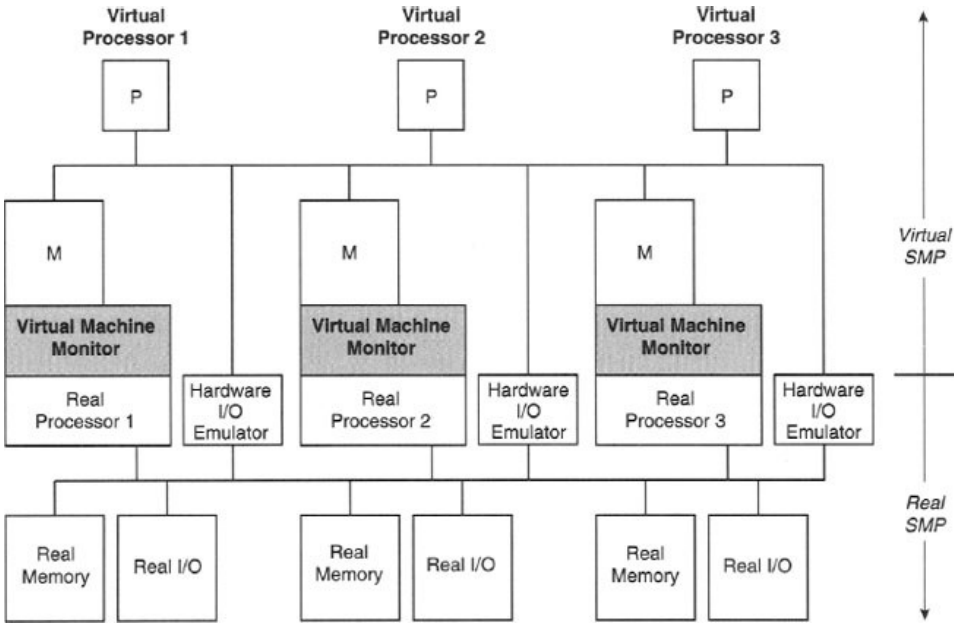


Figure 9.17 Virtual Shared-Memory System on a Real Shared-Memory System. *When there is a mismatch between the I/O of the virtual system and the real system, it may be necessary to add hardware, as shown, for efficient I/O emulation.*

If the host's model does not match the guest's memory model, then it becomes an additional task of the combination of emulation soft-

becomes an additional task of the combination of emulation software running on all processors to communicate with each other and keep track of shared locations. Most solutions to this problem require several instructions and several cycles for each coherence request. Hence, when there is a modest amount of sharing in a system, the emulation performance begins to degrade considerably. The next section examines some ways to handle such cases and discusses the potential degradation of performance. Only a cursory treatment is provided here to this rather complex problem.

9.5.1 Memory Model Emulation

As described in Appendix A, the ISA of most processors includes special conventions for the access of shared-memory locations. As with all aspects of an ISA, these conventions represent a contract between the programmer and an implementation of the ISA. A programmer writes a program assuming that accesses to variables shared among processors obey certain rules, while the hardware designer (the processor designer or the system designer) ensures that the rules used by the programmer are indeed guaranteed by the implementation. We note two main aspects of the memory model, the memory coherence and the memory consistency models, that determine how results appear in different shared-memory systems. These are the two aspects that need to be given special consideration when a multiprocessor system is virtualized—the differences in these models between the host and the guest ISAs can produce undesirable results during virtualization, unless special actions are taken by the emulation software.

Memory Coherence Emulation

Recall that memory coherence is said to be implemented on a multiprocessor system if the order of writes to a given location by one processor is maintained when observed by any other processor in the system. Appendix Section A.7.3 describes the difference in be-

havior of coherent and noncoherent systems. The nature of actions that emulation software must take to support the coherence model of a guest virtual machine depends on the coherence models of both the guest and the host. We list the various possibilities in **Figure 9.18**.

1. **Memory coherence is not required on either the guest or the host:** In this case no action is required because programs have presumably been written in a way that accounts for this absence of coherence. Synchronization in such systems is done through explicit synchronization instructions. The synchronization primitives in the host and the guest may not match, but it is usually not difficult to emulate a synchronization primitive of the guest using synchronization primitives of the host.
2. **Memory coherence is required by the guest and is available on the host:** In this case, the emulation software ensures that the virtual machine runs with the memory coherence option enabled.
3. **Memory coherence is not required on the guest but is enabled on the host:** An example of this is a distributed cluster system that needs to be emulated on a shared-memory multiprocessing system. There is typically no special action that needs to be performed by emulation software. The coherence hardware, which provides a coherent view of memory across the processors, does more than is needed by the coherence model of the guest. If there is a performance disadvantage because of the coherence requirement and there is a faster noncoherent mode available on the machine, this mode can be used, as in case 1 earlier. Synchronization instructions will need to be emulated using the synchronization primitives of the host, even if the host runs in coherent mode.

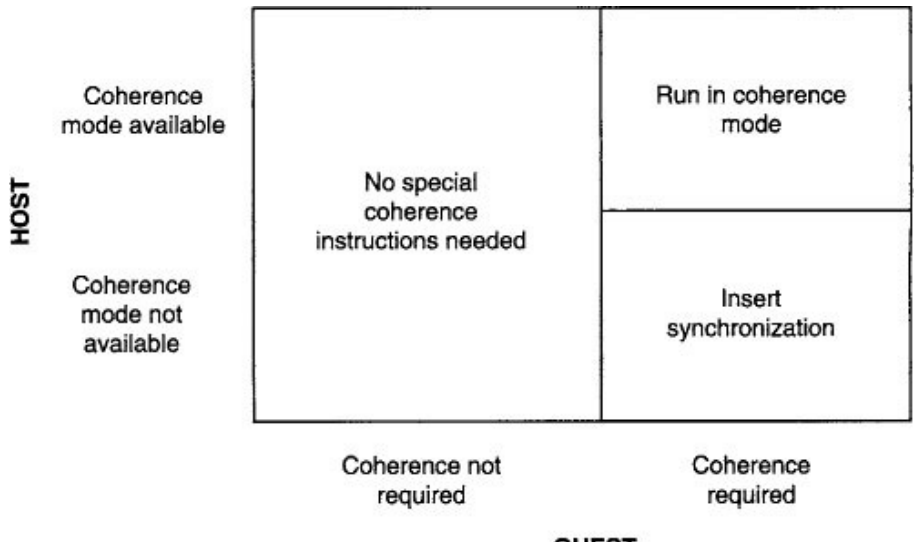




Figure 9.18 The Various Actions Needed for Emulating the Coherence Model. *In all cases synchronization primitives of the guest must be emulated on the host.*

4. **Memory coherence is required on the guest but is not supported by the host:** An example of this is a shared-memory guest machine running on a distributed-memory host machine. The only option here is to use software to enforce coherence. As we said earlier, even when machines do not provide coherence support, they do provide synchronization primitives so that software can perform explicit synchronization. Unfortunately, the brute-force technique of synchronizing after each memory write by a processor is too slow, because synchronization operations, especially on large multi-processor systems, are slow. A more practical approach is to use a directory that registers the set of processors sharing various locations, with a coherence operation being triggered on the set of sharing processors when one of the processors in the set changes the contents of the shared location. This operation can get expensive as the size of the directory gets large. The size of the directory can be reduced by increasing the granularity of an entry in the directory to be a page. Keeping such information at page granularity is also convenient because the available page table structures can be used for storing sharing information and to trigger a coherence action. There have been several techniques of this nature that have been developed in connection with distributed shared-memory (DSM) computing (**Hennessy, Heinrich, and Gupta 1999**), which also deals with the problem of providing coherence to an application while running on noncoherent hardware.

Fortunately most ISAs today provide coherence support. Hence, coherence emulation is not a source of performance degradation for such virtual machines.

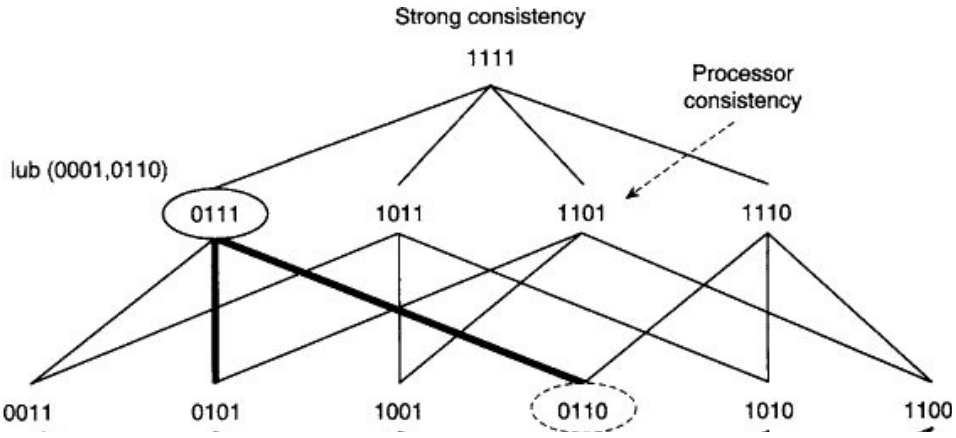
Memory Consistency Models

Memory consistency deals with the order in which accesses by one processor to different locations in memory are observed by another processor (refer to Appendix Section A.7.3 for a more detailed explanation). This is in contrast to memory coherence, which deals with the order of writes to a single location. Memory consistency deals not only with different locations but with all accesses, whether reads or writes. The important thing to take care of when a program is being run on a virtual machine is that the ordering of reads and

writes must be such that it *could have been* produced on a native implementation of the virtual machine. Thus the emulation software should account for differences between the memory consistency models of the host and guest and ensure that the ordering of memory operations is consistent with what the ISA of the guest specifies.

As noted in Appendix Section A.7.3, four hazards have to be considered in a consistency model: the read-read (RR) hazard, the read-write (RW) hazard, the write-read (WR) hazard, and the write-write (WW) hazard. In a strong consistency model, an implementation must ensure that none of these hazards is present. When this requirement is relaxed, we get one of the many relaxed consistency models. Such relaxed models are referred to as being weaker than the strong consistency model.

We could encode the consistency model using four bits, each bit indicating whether ordering is imposed for the corresponding hazard pair. So, for example, a (1111) will indicate strong ordering, while a (1101) will indicate that the WR ordering is relaxed from strong ordering, as in the processor consistency model used on the Intel IA-32 or the IBM System/390. The weakest consistency model (0000) imposes no ordering between any pair of read-write operations. Note that reads and writes to the same location by a processor must be ordered according to program order in all cases (refer to Appendix Section A.7.3).



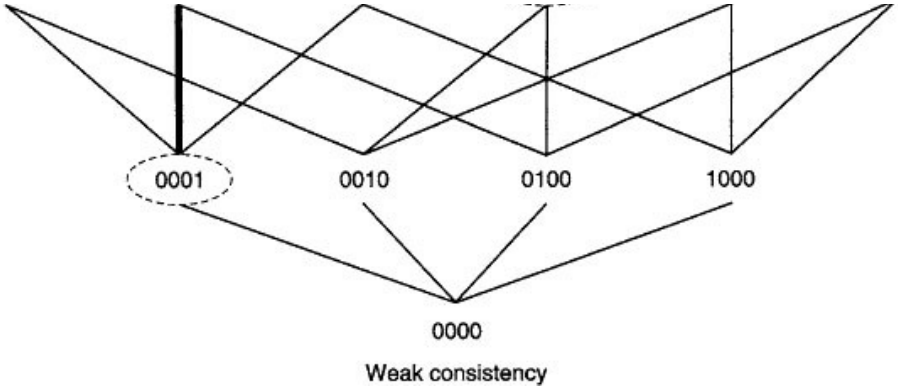


Figure 9.19 Lattice of Consistency Models. *Each 4-tuple represents the four hazards RR, RW, WR, and WW. A "1" in a position indicates that the corresponding hazard is not tolerated, while a "0" indicates that the hazard is ignored by the consistency model. The 4-tuple (0000) represents weak consistency, (1111) represents strong consistency, and (1101) represents processor consistency.*

For purposes of comparing two consistency models, it is convenient to draw a lattice, as shown in **Figure 9.19**, where each node in the lattice corresponds to a consistency model. The least element on the lattice (0000) represents the weakest consistency model, while the greatest element (1111) corresponds to the strongest consistency model. When comparing two consistency models, we determine whether the least upper bound (*lub*) of the pair is itself one of the pair. If it is, then this lub of the pair is defined as being *stronger* than the other. Thus between (1101) and (1001), the lub is (1101), implying that (1101) is stronger than (1001). The model represented by (1111) is stronger than any other consistency model, because it is the greatest element in the set.

As seen from **Figure 9.19**, the nodes (0001) and (0110) are not comparable. One model is stronger in some respects and the other stronger in other respects. There are two models that are stronger than both these models, (0111) and (1111). The lub of the pair is (0111). Any ordering that satisfies the lub (0111) therefore will also be consistent with the ordering specified by the models in the pair (0001) and (0110).

When emulating a consistency model of a guest using a different consistency model for the host, we can divide the space of possibili-

ties into the following three categories.

1. **The memory consistency model of the guest is the same as that of the host or weaker than that of the host:** In this case there is nothing special that needs to be done by the emulation software as long as the interpretation or translation process does not eliminate or reorder memory accesses to shared locations by a processor. This last restriction ensures that the order of accesses to memory in the virtual machine system belongs to the set of orderings of accesses to memory allowed by the consistency model of the virtual machine. Note that this restriction prevents the elimination of memory accesses—an access to a memory location cannot be safely eliminated just because its contents are available in a general-purpose register, for instance.
2. **The memory consistency model of the guest is stronger than (more restrictive than) the memory consistency model of the host:** In this case, it is important that the additional hazards that must be avoided on the guest must also be avoided on the host. We discuss this case in more detail in the next section.
3. **The memory consistency model of the guest is stronger in some respects and weaker in other respects:** In this case we first find the lub between the two consistency models. The lub consistency model must be stronger than the consistency model of the guest. Hence we would be emulating the guest correctly, though conservatively, if we assumed that the consistency model of the guest is the lub model. Since the lub model is also always stronger than the host model, the problem is now reduced to an example of case 2. So, as a fictitious example, if the memory model of the guest is (0110), which relaxes the RR and WR hazards, and the memory model of the host permits all hazards except the WW hazard (0001), then the lub is (0111). Thus in this example, the ordering satisfied by model (0111) would be consistent with the ordering allowed on the guest. Hence it is sufficient to accurately emulate this lub model.

Emulating a Stronger Consistency Guest

When a guest virtual machine has a consistency model that is stronger than that of the host, there are certain hazards that may occur in the host that would violate the memory consistency model of the guest ISA. Let us, for example, consider the case of a guest with a processor consistency model and a host with a release consistency model. This is an interesting case because there are several processors that belong to each of these models—a multiprocessor virtual machine implementation of IA-32 running on a PowerPC multiprocessor would be an example in this category. In this case, it is impor-

cessor would be an example in this category. In this case, it is important that any RR, RW, and WW hazards not permitted in the processor consistency model also not be permitted on the virtual machine.

Most ISAs that implement relaxed forms of consistency also provide special *memory barrier* instructions that permit the programmer to specify the points at which ordering needs to be maintained, hence essentially raising the level of granularity at which consistency is implemented. For purposes of illustration, let us define a *membar* instruction, which, when inserted into the code, ensures that there are no hazards between any pair of instructions across the barrier. Obviously then, a trivial solution for any consistency emulation is for the virtual machine emulator to place a *membar* instruction after every access to memory, as shown in **Figure 9.20**a and b. This ensures that for every pair of memory accesses on any given processor in the host, there is an intervening *membar* instruction that prevents a hazard between the pair. Notice that the same effect could have been obtained by placing a *membar* operation before each memory access.

Unfortunately, the *membar* instruction is an expensive instruction in terms of its execution latency. This can be appreciated better if one understands that the effect of the *membar* instruction is to ensure that all processors in the system have been informed and have acknowledged that some processor wishes to impose such an ordering. Each processor then takes appropriate action to ensure that the request is honored. Since loads and stores form roughly a third of all instructions in an instruction stream, the performance of the virtual machine can be degraded quite significantly because of the insertion of these barrier instructions. It is therefore important for the monitor to restrict the number of *membar* instructions whenever possible.

Notice that the trivial solution just demonstrated works for any consistency model on the guest and, in particular, for the strong model.

The first optimization that can be performed is to eliminate those memory barrier instructions where the memory model allows relaxation. In the case of processor consistency, the memory barrier instruction after a write can be eliminated as shown in **Figure 9.20**c if the subsequent memory access is by a read instruction. The second optimization that can be performed is to eliminate a memory barrier instruction after an instruction if the next sequential memory access is to the same location as the previous memory access—program order requirements take care of these even without barrier instructions. This must, however, be done with care. In the example of **Figure 9.20**c the *membar* instruction following Read A cannot be eliminated because the *membar* instruction just after Write A has been removed, thus permitting the following Read B to be done before Write A. However, processor consistency does not permit Read B to be done before Read A, implying that there must be at least one *membar* between these two instructions. Notice that the *membar* that was between the Read C—Write C can be eliminated without danger of the Write B moving ahead. It is important to note that there are other ways of inserting *membars* in the code that achieve the same objective.

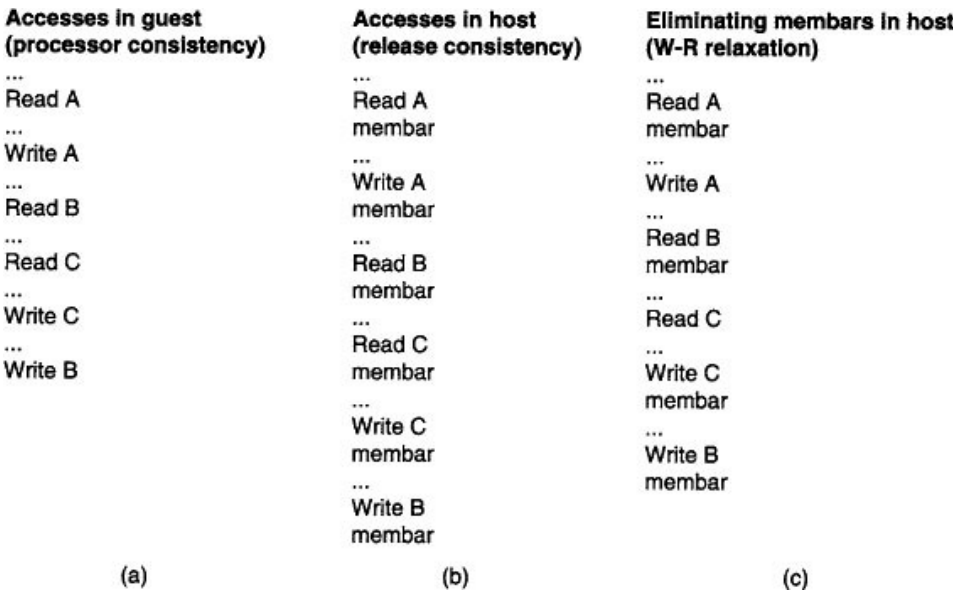


Figure 9.20 Using *membar* Instructions to Impose Consistency Requirements of the Guest That Are Stricter Than That of the Host. *(a) Original host code. (b) Conservative use of membar instructions after each memory access. (c) Reducing the number of membars to account for W-R relaxation in the guest.*

count for W-R relaxation in the guest.

This example shows that it is difficult to reduce the overhead of inserting memory barrier instructions when a guest has a stronger consistency model than the host. However, there is another way to reduce the overhead of such emulation. Even in a multiprocessing system there are many occasions when a processor executes a single threaded application or a portion of a multithreaded application that does not share memory with other threads. Since the hazards we are trying to avoid are multiprocessor hazards, there is no need to add barrier instructions if there is some way for the virtual machine monitor to know that a thread does not share memory with an active thread on another processor. The translator in such systems is free to move memory accesses relative to each other, as long as it provides a way to recover the state of the virtual machine processor whenever it is needed.

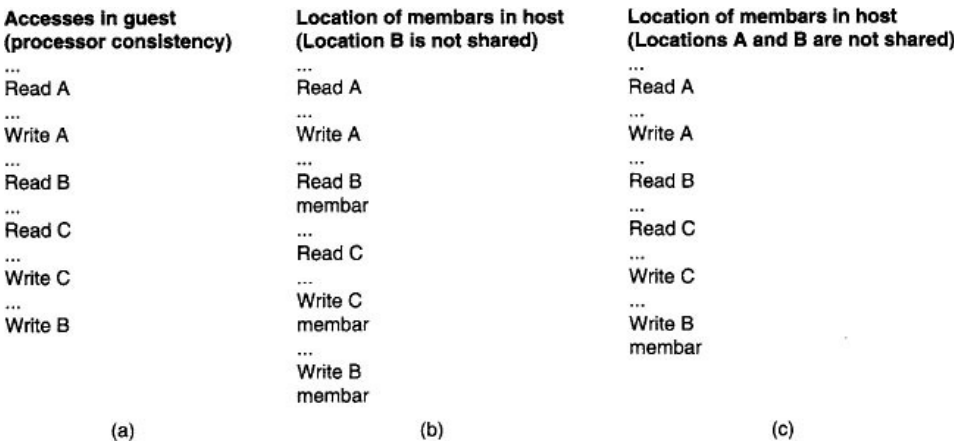


Figure 9.21 Reducing the Number of Memory Barrier Instructions Using Knowledge About Sharing Characteristics of Access Locations. *We leave the last membar in the segment, because at this point we have no knowledge of the sharing pattern of the code that follows the sample segment.*

A more interesting situation arises when an application is written in such a way that the threads comprising the application are largely independent and only occasionally share memory, perhaps as a form of communication. In these cases, it may be more efficient to restrict the insertion of memory barriers to keep the required ordering of memory access to only the shared locations, as shown in

Figure 9.21.

The figure shows that when location B is not shared, we can remove the barriers between Read A and Read B because any reordering of access to location B will not be visible to any other processor. If neither location A nor location B is shared, then the only ordering visible to other processors is the Read C–Write C ordering. No *membar* is needed between this pair because of program order requirements. Hence it is unnecessary to place a *membar* anywhere in the code.

Thus the principal obstacle to good performance is the problem of determining efficiently during translation whether an access is to a shared location or not. This is a difficult problem because, although the history of accesses by an instruction can provide some measure of confidence about whether or not that access is to a shared location, there is a lot of bookkeeping involved in determining whether locations are shared, and extra overhead involved in ensuring that a trap is generated when a previously unshared location becomes shared.

Another problem is that of *false sharing*. False sharing occurs, for example, when the emulation software detects an access to a location by two different virtual processors but does not know that the guest operating system has migrated the thread from one processor to another for load-balancing purposes. False sharing could therefore lead to the elimination of fewer memory barriers than what would be possible.

Real ISAs often allow other relaxations or impose other orderings beyond the ones mentioned here. For example, on the System/390, a distinction is made between instruction reads and data reads. The ISA does not require that instructions be fetched in the order they are eventually executed. It also allows instruction fetches to occur before the reading of operands of previous instructions. Such relax-

ations, which obviously arise from a desire to make hardware more efficient, also help in reducing the overhead of emulation.

We have seen in this section that creating a virtual machine system with an ISA different from the ISA of the native host is certainly possible, though the efficiency of such virtualization depends a lot on the mismatch between the memory models of the two ISAs. In particular, when the memory model of the host is more relaxed, due either to a weaker coherence model or a weaker consistency model, the potential performance degradation is high, unless the emulation software performs more work in localizing the coherence and synchronization mismatches. As with VM assists mentioned in **Chapter 8**, hardware could be added to assist the emulation software in this task. An extreme example of this is for the processor to provide a mode in which it supports the stronger memory models required by the guests. There are processor ISAs, such as the PowerPC ISA, that provide both a coherence mode and a mode in which coherence may not be required. There are also processor ISAs, such as Sun's SPARC ISA, that support multiple consistency models.

9.6 Summary

Large companies and other organizations are continually grappling with the problem of reducing the number and types of computer systems they must maintain without depriving their important users of the flexibility of maintaining and operating their own system. This has led to an array of systems claimed to enable such consolidation, basically by decoupling the view of the hardware as seen by the operating system from the actual hardware that the system runs on. Virtual platforms on large servers provide users with characteristics similar to those they would have enjoyed on their own private physical systems.

This chapter examined various ways in which large multiprocessor

servers can be adapted to provide multiple virtual multiprocessor platforms to various users. We have seen how different forms of virtualization handle the task of isolating virtual machines from each other while still providing the benefits of sharing resources. Some of these solutions, such as physical partitioning, are superior at isolating virtual machines from hardware failures, while others, especially the codesigned hardware–software approaches, excel in the degree of flexibility they provide in the allocation of resources to the virtual machines.

The biggest impediment to the adoption of new instruction set architectures is the existence of large bodies of application programs in binary form for the existing ISAs. Large multiprocessor systems of today are usually homogeneous—they incorporate processors that all execute the same ISA in their native mode. For most organizations, consolidating to a large multiprocessing system with one ISA will still leave a significant number of important applications in platforms of other ISAs that need to be supported, at least until they are ported to the native ISA. Hence virtualization of these other platforms will remain an important subject, at least until a more widespread adoption of platform-independent languages, such as Java, and platform-independent operating systems, such as Linux. We have seen that efficiently emulating applications of one ISA on another, while already difficult for uniprocessor platforms, is quite daunting in the case of multiprocessor systems, especially when there are incompatibilities in the memory consistency model. It is likely that new techniques will be developed, particularly exploiting thread-level parallelism, to improve the performance of emulating one ISA on another.

Same-ISA virtualization, on the other hand, appears to be here to stay. While the advantages of virtualization listed in the introduction to this chapter are compelling for enterprise servers, it is likely that smaller servers, such as those of Internet service providers, will

also begin to employ virtualization, particularly as a means of ensuring privacy between users and as a means of protecting users from malicious or accidental failures of the platform employed by other users. Indeed, as more applications use network resources for their execution, the need to protect one application from another, even in a single-user environment, may eventually lead to the adoption of virtualization techniques on personal devices such as laptop computers or even mobile phones.

---

1. Note that the Cellular Disco designers refer to the real memory as the physical memory of the virtual machine and to the actually available physical memory of the system as the machine memory.

Previous chapter
Chapter 8: System Virtual Machines

Next chapter
Chapter 10: Emerging Applications

Chapter Ten

Emerging Applications

With ever-expanding computer applications and their associated demands on computer system administration, the role of virtual machines and virtualization are likewise expanding. Virtualization enables the introduction of new system capabilities without adding complexity to already-existing and already-complex hardware and software. These new capabilities will support innovative computing paradigms and will provide critical functions in areas that have assumed major importance in a world grown increasingly dependent on computers.

In an argument reminiscent of those given for the original development of VM systems on mainframes, it has been suggested that operating systems and applications running on real machines today should be relocated to virtual machines. Part of the motivation for this is the desire to allow new services to be developed in a layer between the hardware and the operating system without modifying or trusting either the applications or the operating system. Examples of new services are intrusion-detection systems that help in isolating users from malicious attacks, environment-migration services that help mobile users access their entire computing environment without having to carry their machines everywhere, and enhanced reliability through redundant execution of a program on multiple virtual machines. There have been several efforts that use virtual machine techniques to emulate one ISA on another. Virtual machines are being used to package entire environments and applications to avoid the time-consuming job of installing operating systems and applications on machines. They are also being used to isolate data on a system, thereby allowing the same physical machine to work with confidential and public data without compromising security.

fidential and public data without compromising security.

To illustrate the importance of virtual machine technologies to future systems and paradigms, in this chapter we describe three examples of emerging virtual machine applications that can make a fundamental difference in the future of computing. The first of these is the application of virtual machines to computer security, an area receiving increasing attention. Modern computing systems are some of the most complex structures ever created. With the commoditization of both hardware and software and with widespread use of network computing, it has become easier for intruders to cause damage on a large scale by taking advantage of the nearly inevitable loopholes and flaws in such complex, constantly evolving systems. Eventually, systems will mature and become more immune to attack, and it is likely that integrated mechanisms will be developed specifically aimed at thwarting such attacks. The virtual machine intrusion-detection mechanisms described in this chapter promise to be useful tools in the development of such attack-immune systems.

The second application we consider, the migration of complete computing environments, is also motivated by the commoditization and networking of computing systems. A growing number of computer users depend on having their full environment, including all data, programs, and system personalization, readily available at multiple locations (e.g., at work, at home, and while traveling). In order to do this, however, these users must carry their environment with them on a portable machine. It is more desirable, however, both for convenience as well as for security, to allow a user to log in to any computer and immediately reproduce the user's environment on that specific computer. Virtual machine technology, which provides the ability to capture the entire state of a computer system, facilitates migration of a full computing environment. Eventually, users may become accustomed to having their data and computing environment on large, remote computer farms. While such a paradigm may

obviate the need for having an entire computer at each location to which the user travels, it will not eliminate the need for migrating entire user environments from one system to another. Load-balancing, system-utilization, and system-latency issues will require means for capturing the state of the user environment and making it portable.

The notion of computing as a service or as a utility is beginning to gain acceptance. The scientific community has been actively promoting this paradigm for some time, and it also appears to be a concept that will be useful for businesses. The concept of a computational *grid* promises a seamless way through which computing resources can be shared by multitudes of users. The grid virtualizes computing resources in a way that is philosophically similar to the virtualization of resources in system virtual machines, though technically it achieves virtualization by redefining the way applications are written. In a sense, the grid concept represents the ultimate evolution of virtual machines.

Just as the Java platform fundamentally embodies virtualization, portability, and safety in its definition, the grid fundamentally embodies these same characteristics at the system and application levels. Virtualization is not optional on the grid—applications running on the grid must accept the notion that the physical characteristics of the real resources they are using are unknown to them. The hardware and operating system of the individual machines have the responsibility for providing this implementation-independent view. Although today this is done by loading extensions on existing systems, it is likely that, as in the case of security features mentioned earlier, support for operation on a grid will be integrated with the design of machine hardware and basic system software. This integration will likely include many of the virtual machine concepts described in this book.

We begin our survey of emerging VM applications in **Section 10.1**

with a discussion of the role that virtual machines can play in maintaining security in our increasingly vulnerable computer systems. In **Section 10.2**, we examine the idea of virtual machines as the enabling technology for migrating entire environments from one computer system to another. Finally, in **Section 10.3**, we take a look at the emergence of the grid computing paradigm, its similarities to other virtualization techniques, and its potential role in making virtual machines ubiquitous.

10.1 Security

Users of commercial systems, especially mainframe computers, have long been concerned with system security. These machines, which are used by large institutions, including banks, airlines, and federal agencies, handle sensitive information and are required to be secure against attacks. Security has not been as much of a concern, however, for users of common commodity computers—until relatively recently. Today, with the increase in the number of security attacks as well as the increasing cleverness of the perpetrators, computer security has become a major issue in all types of computing, including low-end desktop machines, portable machines, and even personal digital assistants (PDAs) and cellular phones. Seldom does a month pass when we do not hear about (or, even worse, fall victim to) yet another worm or virus that has invaded many thousands (if not millions) of computers around the world.

The most common way the security of a system is compromised is simply for an attacker to get access to the privileged part of the system, e.g., as a superuser in Unix environments or as an administrator in Windows systems. Interestingly, a large number of attacks are carried out by people who have gained access to such privileges without much effort—for example, through inadequately protected passwords or, more commonly, through repeated attempts to gain access to the system via commonly used passwords. Once the at-

access to the system via commonly used passwords. Once the attacker gains access to the system, he or she can tamper with the operating system to gain confidential information or even to destroy data. Often, during the initial entry into a system, an intruder simply modifies the system to ease entry into the system in the future.

Another common type of attack exploits inherent weaknesses in system software. The size and complexity of modern system software, especially the operating system, makes it very difficult to ensure that every security hole is eliminated before the system is shipped to customers. Programs written in languages such as C, which are considered "unsafe," often exhibit bugs (or possess security holes resulting from poor programming practices) that can be exploited to gain access and hence control the system. One common type of security hole involves unchecked accesses to C arrays. When such an array is used as a buffer for user data, the user can fill the buffer with a block of data that is bigger than the array size and cause memory areas adjacent to the buffer to be overwritten. If one of these overwritten areas happens to be an address to which the program jumps, an attacker, through clever calculation, can cause that address to point to a routine, e.g., a shell program, by which the attacker gains control of the system. This is illustrated in **Figure 10.1**. It is generally not likely for an innocent user inadvertently to do harm to the system in this manner because the probability that arbitrary user data will cause a jump to a sensitive location is low. However, such security holes can be exploited maliciously and cause havoc in systems.

10.1.1 Intrusion-Detection Systems

In practice, security holes due to bugs or poor programming, as described in the previous section, eventually get fixed. However, as new code is introduced to a system, new holes appear. Clearly the most effective measure one can take to ensure the safety of a system is to isolate the system from all potential attackers (which is often

the same as isolating it from all other systems). This is not a practical solution for most systems—users are dependent on communication with other systems either over local-area networks or over the Internet, and this makes them vulnerable to malicious attacks of the kind mentioned earlier. Using object-oriented programming with built-in type and range checking as is done in Java and MSIL will also provide a high level of security, but only for those programs running within the HLL VM framework. A significant amount of system code and a number of native libraries are likely to remain outside the secure framework for quite some time.

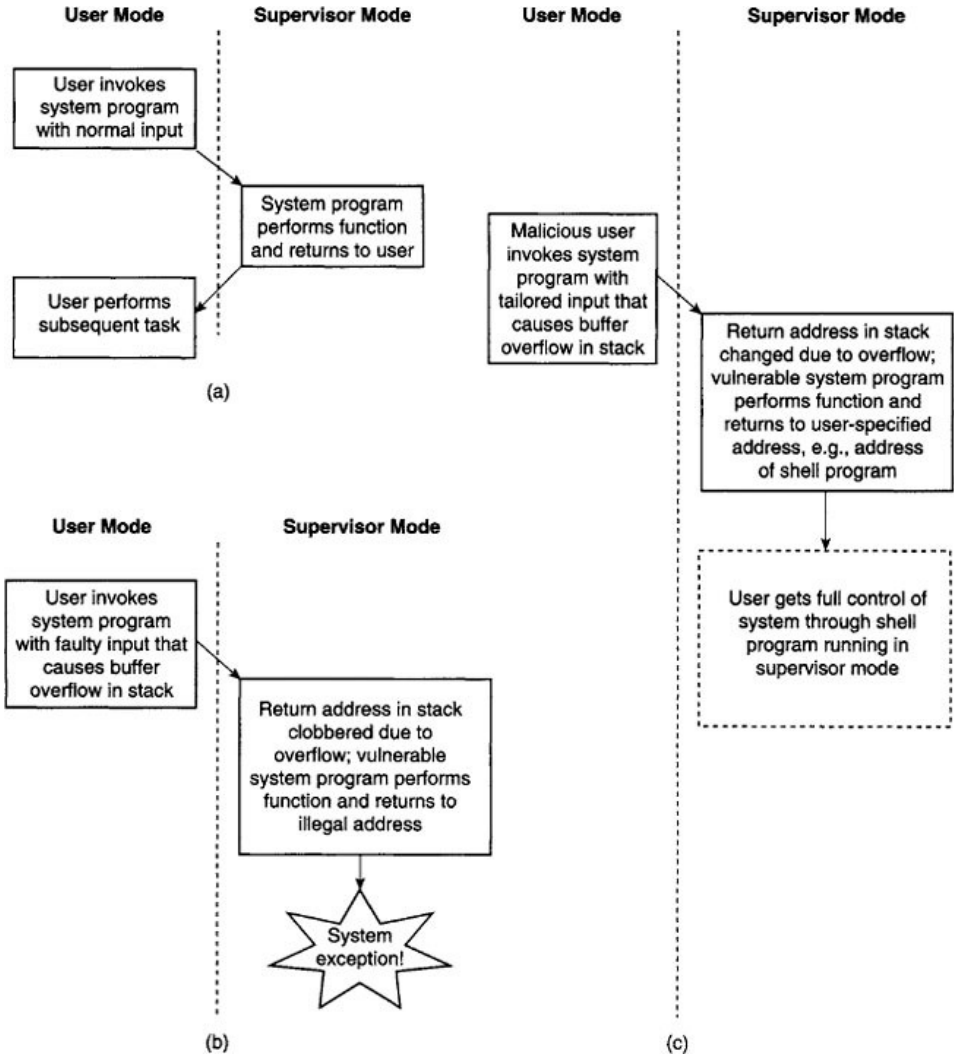


Figure 10.1 Exploiting Buffer Overflow in a Program Stack. *In the normal course of events (a) the user parameter fits in the buffer. The return address on a stack is that of a location in the user's program. If the user inadvertently causes an overflow of the buffer (b), the return address may be clobbered (overwritten) but typically not with a legal instruction address. In this case, the return causes an exception. A malicious program (c) can manipulate*

the parameter to be copied into the buffer so that it overwrites the return address with a location that allows the adversary to launch another valid program. This program can be a shell program that, now running in the supervisor mode, allows the user free rein over the system.

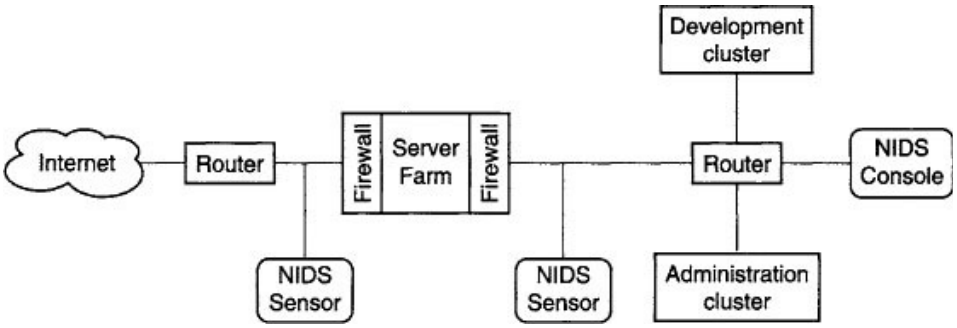


Figure 10.2 Typical Enterprise Network Intrusion-Detection System (NIDS). *The sensors monitor the network for spurious activity. The console allows the programming of the sensors, logs the activity seen by the sensors, and forms the control point to isolate parts of the system when malicious activity is detected.*

A common way of providing security protection for current systems is through the use of an *intrusion-detection system* (IDS). As the name suggests, these systems attempt to examine a computer either continuously or periodically to check for potential attacks. These systems depend on the general knowledge of how potential attacks may take place. Once the characteristics of an attack are known, an IDS can examine the activity on a computer to determine whether an attack with the given characteristics is occurring and then take action to shut down any malicious processes before they can cause significant harm. Since most attacks are launched from the network to which a system is connected, an intrusion system can be located at some point in the network away from the computer—in this case it is termed a *network-based intrusion-detection system,* or NIDS for short. If, on the other hand, the intrusion system is located within the computer itself, for example, in the operating system, it is referred to as a *host-based intrusion-detection system,* or HIDS.

Network-based intrusion systems (**Figure 10.2**) work by examining data packets as they move through the network, looking for specific patterns, or *signatures,* of suspicious activities. The traffic can be monitored either at the host or at points more remote from the host, e.g., at firewalls or routers. As soon as suspicious activity is detected,

a NIDS can block the activity or reroute the traffic to a quarantine location. Since it is possible for attacks to be disguised, there is no guarantee that malicious activity will always match one of the signatures in the NIDS database. There is also the possibility of innocuous traffic being deemed suspicious because of a similarity in its signature to one in the database (i.e., a *false positive*). Still, the NIDS has proven to be quite effective in many situations, particularly because it is decoupled from the system being protected.

Host-based intrusion-detection systems work by directly examining activity on the host and taking advantage of detailed knowledge of the host's operating system. A HIDS will look, for example, for repeated attempts to login by an intruder, or it will look for attempted accesses to files that should normally not be accessed by a user. Typically, the HIDS is integrated with the operating system and has high visibility over all the actions that take place on the system. Even the HIDS that operates as an application on the host enjoys high visibility by interacting closely with the operating system. In fact, such decoupling of the HIDS from the operating system can provide more protection to the system from intrusions that attempt to attack the HIDS itself.

Thus it is clear that the two types of systems are different in several respects and have their own advantages and disadvantages. A network-based intrusion system has significantly less visibility than a HIDS and has to base its decisions almost entirely on an examination of network traffic. An attacker thus has more flexibility in maneuvering around the detection system, especially by camouflaging itself. On the other hand, the NIDS can continue to monitor activity even after a host system has been successfully attacked—this is invaluable in learning about the nature of the attack and in taking measures to prevent it in the future. A system incorporating a HIDS can become completely disoriented after an attack, perhaps even providing misleading information while giving the intruder enough time to marshal additional resources to gain control of the system.

10.1.2 Monitoring and Recovering from Attacks

Attackers are constantly finding new ways to get around the defenses of a system. Thus while it is important for systems to have good intrusion-detection systems, it is equally important to take adequate measures to recover from an attack and to prevent similar attacks in the future. One important part of this effort is to perform *logging*. Logging saves information about critical activity on a system, e.g., login attempts and accesses or changes to certain important files on the system. This type of limited information can help in the analysis of methods used by an intruder and hence help in developing defenses for future attacks. However, in order to be able to recover fully from an attack, it is necessary to save much more information, for example, a checkpoint of the state of the system at some point before an attack occurs.

Many techniques and policies have been proposed for logging of events in a system. Most of these techniques assume that the operating system is functioning correctly, because they use the services of the operating system in order to record various events that occur. In fact many of them even keep the log of activities on the system being monitored. Such systems are doubly compromised. When an attack takes place, not only does the system fall under the control of the attacker, but the log of the system may also be altered in a way that is transparent to the user of the system or in a way that prevents reconstructing the attack for building up new defenses.

There are also difficulties involved with the reconstruction of a system after an attack. The first requirement is determining a known good state of the system before the occurrence of the attack. This implies that a checkpoint of the system must be available for some point in the past. From this state the actions taken by the system can

be replayed up to some later point, if complete knowledge were available about all the inputs into the system following the checkpoint. If the times at which these external inputs, such as keyboard activity or network activity were known, the system could be brought back step by step to a point just before the occurrence of the suspicious activity. Unfortunately, not all external input events result in deterministic changes to the system. Asynchronous interrupts occurring because of events outside the core, for example, from an I/O device, may not occur spontaneously or may not occur at exactly the same point when the system is replayed. Such events must be logged in sufficient detail so that they can be simulated accurately when the system is being replayed.

This completes a brief overview of the typical ways in which systems are attacked and of the approaches used by intrusion-detection systems to detect attacks. We have also outlined the requirements of system logs in order for a system to be able to recover from an attack. In the next section we examine how virtual machine technology facilitates these functions.

10.1.3 Role of Virtual Machine Technology

Most of the techniques needed to maintain security in a system have usually been implemented as separate programs on existing systems. It is reasonable to believe there would be significant advantages if the security features were integrated closely with the rest of the system. Yet it is important that techniques used to enforce security and monitor activity on a system be implemented in a way that separates their implementation from that of the system being attacked. This is where virtual machine technology comes in. We will look at three examples of the use of virtual machines in the area of system security.

Virtual Machine as a Sandbox

We have seen in **Chapters 8** and **9** how virtual machines allow the isolation of complete system environments from one another. Fault containment is an important property of virtual machines—a software failure on one virtual machine does not generally propagate to other virtual machines. Typical virtual machine systems are capable of isolating the failed virtual machine, shutting it down, and then restarting a new virtual machine without these actions being noticed by the other virtual machines in the system. This ability to isolate a virtual machine makes it useful as a tool (**Figure 10.3**) for closely examining the effects of an attack after it has occurred. As noted earlier, such postmortem analysis is useful in designing a defense against future attacks or even toward detecting such attacks before they cause harm.

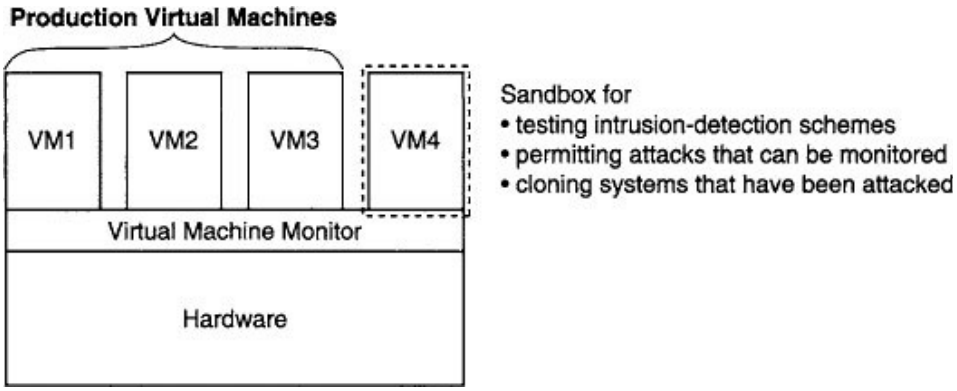


Figure 10.3 A System Virtual Machine as a Sandbox.

In a paper describing the advantages of virtual machines, Chen and Noble suggest that the complete state of a virtual machine can be saved, cloned, encrypted, moved, or restored—actions that are not so easy to do with physical machines (**P. M. Chen and Noble 2001**). They further suggest that the best way to understand the effects of an attack is to replay the attack and gather information about the behavior of the attack by monitoring the system as the attack progresses. While this is risky to do on a real system, one could safely carry this out on a virtual machine by cloning the original machine that was attacked, inspecting the effects of the attack, and throwing away the machine after it has served its purpose. (Cloning of virtual

machines can be done using the system encapsulation and restart techniques described in **Section 10.2**.)

Cloning a system on a virtual machine opens up other interesting possibilities. A potentially suspicious network packet or other suspicious input can be sent to a clone before it is forwarded to the actual system, to see if it has any ill effects. Similarly, during the development of intrusion-detection systems, tests for the effectiveness of the system can be conducted more safely on a cloned system running on a virtual machine than directly on the system that needs to be protected.

Virtual Machine for Monitoring Low-Level Activity

A system virtual machine forms a barrier between a system and the hardware on which it is running and also between a system and other systems running on the same hardware. This property of the virtual machine can be extended to provide a barrier between a system potentially under attack by intruders and the IDS that monitors the low-level activity of the system. Here are two possible ways to configure an IDS in a virtual machine system to achieve this.

1. The IDS can be written as a separate process in its own virtual machine, or on the host operating system of a hosted virtual machine, with special access to the functions of the VMM through a dedicated interface. This interface must support (a) means to send commands from the IDS to the VMM, for example, to enable certain monitoring functions, (b) a mechanism to efficiently access the physical memory of the system being protected, and (c) means for the VMM to transmit information about activity on the virtual machine back to the IDS.
2. The IDS can be integrated with the VMM. This gives the IDS the privilege and status of the virtual machine monitor and allows it complete access to all hardware activity of the system being protected. The VMM is a critical, central piece of software in a system and should therefore be thoroughly debugged and hardened against attacks and failures. In a scheme where the IDS is integrated with the VMM, it is imperative for the IDS also to be written, debugged, and verified with the same rigor.

Separating the IDS from the VMM makes the functioning of the IDS a

bit less efficient because of the need to communicate through an interface, as mentioned earlier. However, it avoids the addition of baggage to the VMM, a component that is central to the operation of the system and hence that must be efficient and verifiable.

The Livewire system (**Garfinkel and Rosenblum 2003**) is an example of a system that separates the IDS from the VMM. The changes needed to a VMM for supporting the interface required by an external IDS are minimal and need not compromise the robustness and efficiency of the VMM itself. The IDS configures the VMM to gather information about the activity of a virtual machine. To enable this, the system specifies additional events or instructions at which the virtual machine relinquishes control to the VMM so that the required information can be gathered. The VMM is also provided with a list of checks to be made on the gathered information. These checks may be a simple matter of matching the activity against templates of known malicious activity. Thus, once set up, the VMM performs all the needed activity without communicating back and forth with the IDS module. Once the signature of a malicious event is detected, the virtual machine is suspended, and the IDS is signaled and provided with the monitoring data gathered. The IDS examines the data and either responds with a command instructing the VMM to resume operation of the virtual machine or halts the virtual machine if it cannot guarantee that the system has not been compromised.

The system just described is simple because it is conceptually stateless. The decision about whether to halt a virtual machine or to allow it to continue is based only on the information provided by the VMM when suspicious activity is detected. This may be restrictive. Certain security measures require the incorporation of state in the intrusion-detection mechanism. Often, a suspicious event may not be directly identifiable as malicious but may require further examination of specific other events in order to be identified as such. For

example, monitoring the system call activity immediately after a series of attempts to log in could give greater confidence in identifying the user logging in as an intruder. To achieve this, the IDS can send a command to the VMM immediately after a series of unsuccessful login attempts, instructing it to monitor all traps and interrupts intended for the virtual machine being protected.

Unlike the VMM, an IDS needs to have information about the type of operating system running on the virtual machine. The characteristics of intrusions often tend to be specific to the operating systems being attacked. As a result, an IDS tends to have several operating system—specific mechanisms. The Livewire system, for example, includes an OS interface library that allows the IDS to perform an analysis on a crash dump generated by a virtual machine, in almost an identical way as it would have been done by an application running on the operating system.

We mentioned earlier that the VMM is provided with a list of checks and templates to be used during the monitoring process. This is the function of the *policy engine,* the part of the IDS that determines how the monitoring should progress and whether the observed behavior matches the signature of malicious activity. The interface to the VMM, along with the mentioned OS interface library, provides a framework on which to build the policy modules of the policy engine. Livewire (see **Figure 10.4**) used this technique to build several policy modules, each of which was only between 30 and 130 lines of code. More details about the implementation of these for a Linux-based system are presented in **Garfinkel and Rosenblum (2003)**; following are a few examples.

Lie detector module: Attackers often want to mask their activity by lying about the system. They do this by getting control of the system, modifying it, and then masking their activity by providing false responses to system queries. A VM-based intrusion-detection system can thwart such attempts by sending an enquiry to the attacked sys-

tem about the state of the system and comparing the results it provides with the actual state of the hardware of the system, which is completely known to the VMM.

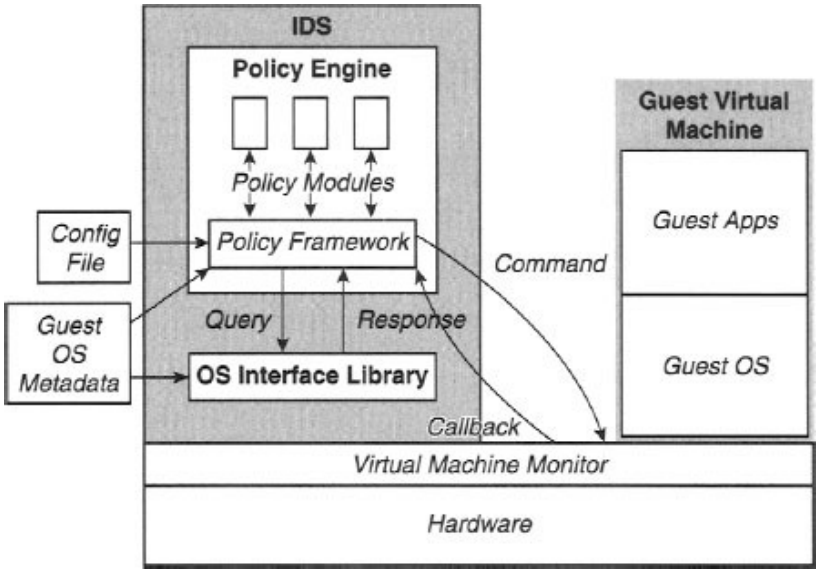


Figure 10.4 Livewire Intrusion-Detection System (Garfinkel and Rosenblum 2003). *The IDS may be viewed as another virtual machine built on top of the VMM but with special privileges. The major components of the IDS are (a) the OS interface library, which provides an OS-level view of the monitored virtual machine, obtained by interpreting the metadata returned by the guest OS through the VMM, and (b) the policy engine, which provides a framework to implement common policies as well as a set of policy modules each of which implements an intrusion-detection scheme. The VMM isolates the IDS from the monitored virtual machine.*

User program integrity detector module: Checking the binary stored on disk is not helpful after an attacker has modified the contents of a long-running program in memory. Livewire uses the VMM to check every code page of a program in memory and generates a signature (e.g., checksum) that is then compared with known good signatures for the same pages saved elsewhere. Any discrepancy is flagged as a potential violation of security.

Signature detector module: Antivirus programs have traditionally relied on the fact that most new viruses and Trojan horse programs simply reuse known techniques—often just copying the basic kernel code of an old virus or Trojan horse. Thus it is often effective to create a signature of these kernels and to check for this signature in files that are likely to be attacked. Virtual machine technology allows such a detection technique to be extended to the entire mem

lows such a detection technique to be extended to the entire memory of the system. The VMM has access to the virtual hardware of the virtual machine, and hence an intrusion-detection system can use the interface of the VMM to schedule a periodic scan of the memory for such signatures.

Memory access enforcer module: Traditional operating systems protect code sections and other important parts of memory by rendering them read-only. Integrity check techniques are sometimes employed to ensure that certain critical sections of memory have not been tampered with and are still protected as read-only. But these techniques can detect such attacks only after they have occurred. By running the system inside a virtual machine, it is possible to get better and more immediate protection from such attacks, by rendering the same critical pages as read-only. Any attempt to change the access privilege of these pages is intercepted by the VMM, which can then halt the system.

The implemented policy modules generally fall into two categories: *polling modules* and *event-driven modules.* Polling modules poll the system at regular intervals to look for malicious activity, while event-driven modules report that some undesirable event has occurred on the system. The memory-access enforcer is an example of an event-driven policy module; the others are examples of polling modules.

Virtual machine technology helps eliminate some of the disadvantages that a HIDS has in comparison to the NIDS, with its better isolation characteristics. While it is natural to extend existing virtual machine systems to support intrusion detection, it is also possible that intrusion detection may one day be the reason for deployment of virtual machines.

Secure and Complete Logging Using Virtual Machines

Logging of activity on a system is an important part in the fight

Logging of activity on a system is an important part in the fight against malicious attacks on a computer system. As mentioned earlier in this chapter, a log of the activity on a system enables analysis of the events associated with an attack, especially if such data is available from the period both immediately prior to the attack and as the attack progresses. Such an analysis could lead to techniques that (a) anticipate an attack and (b) know when an attack has occurred.

A common method of logging is to record all accesses to important and critical parts of the system, e.g., login attempts, network-related events, access to the system registry, and commands invoked at the superuser or system-administrator level. Unfortunately this information is not trustworthy when under attack—an attacker gaining control of the operating system can change the logging information to mask the attack. Moreover, while evidence of the intrusion may be available from the log, there may be insufficient information to determine the exact vulnerability that led to the success of the attack.

Researchers at the University of Michigan (**Dunlap et al. 2002**) have proposed a virtual machine—based system called ReVirt that attempts to address both these problems, namely, the inability to guarantee the integrity of logging information as well as the incompleteness of traditional logging information. They use the VMM both to separate the logging process from the system being monitored and to collect all information needed to replay instruction-by-instruction system activity for a long period of time—potentially starting from a point before the start of the attack and continuing to well after the attack is under way.

Any computing system may be viewed as a finite-state machine. If the starting state is known, it is sufficient to know only the sequence of inputs to be able to replay its entire execution; the system moves through a set of states that can generally be determined simply by

re-executing the program. However, there are situations where the transition between the states of a system is time dependent—reproducing the effects of an earlier execution in such cases requires a simulation of the input at the time recorded during the earlier execution.

A better reference for recording nondeterministic events is the number of instructions that have been completed by the machine rather than the time on the system clock. It has been observed (**Bressoud and Schneider 1996**) that even when the inputs are completely deterministic, for example, in a program with no I/O, the time at which an instruction is executed varies from run to run, even though the count of instructions executed remains the same. This is due to a number of factors, the most obvious one being the variable latency to cache memory, depending on the interaction with other applications simultaneously running on the system. Modern processors often provide performance-monitoring hardware to count completed instructions, and hence it may be feasible to trigger events at specified instruction execution counts. However, a more efficient and an equally effective measure is the number of branches that have been executed by the machine. The location of an event can be completely characterized by the exact number of branches that have been executed from the program start, along with the value of the instruction program counter when the event occurs.

Counting events, e.g., the number of instructions executed or the number of branches executed, is accomplished through the setting of special performance counters visible at the ISA level. Each time the event occurs, the counter counts down by 1. When the counter reaches a value of zero, an interrupt can be scheduled to inform the application that the desired count has been reached. The designers of ReVirt observed that on IA-32 processors (and possibly on other processors too), the interrupt generated by the counter does not stop execution immediately and could in fact occur several tens of in-

structions later. To ensure the delivery of a nondeterministic event at a precise point in the computation, they use a two-phase technique, shown in **Figure 10.5**. At the start of the replay the precise number of branches to be executed before the occurrence of the event is known. The instruction address of the desired point within the basic block after the last branch is also known. In the first phase, the branch performance counter of the IA-32 processor is set so that 128 branches will still remain to be executed. The interrupt from the counter is received at some point where the number of branches remaining to be executed is less than 128. At this point, the second-phase strategy is invoked. In this phase, a breakpoint is set to the program counter value recorded for the event. Each time the execution arrives at the breakpoint, the number of branches is examined to determine whether the desired number of branches have been executed. The exact point is reached when the desired number of branches have been executed, and execution progresses to the instruction at the desired instruction address.

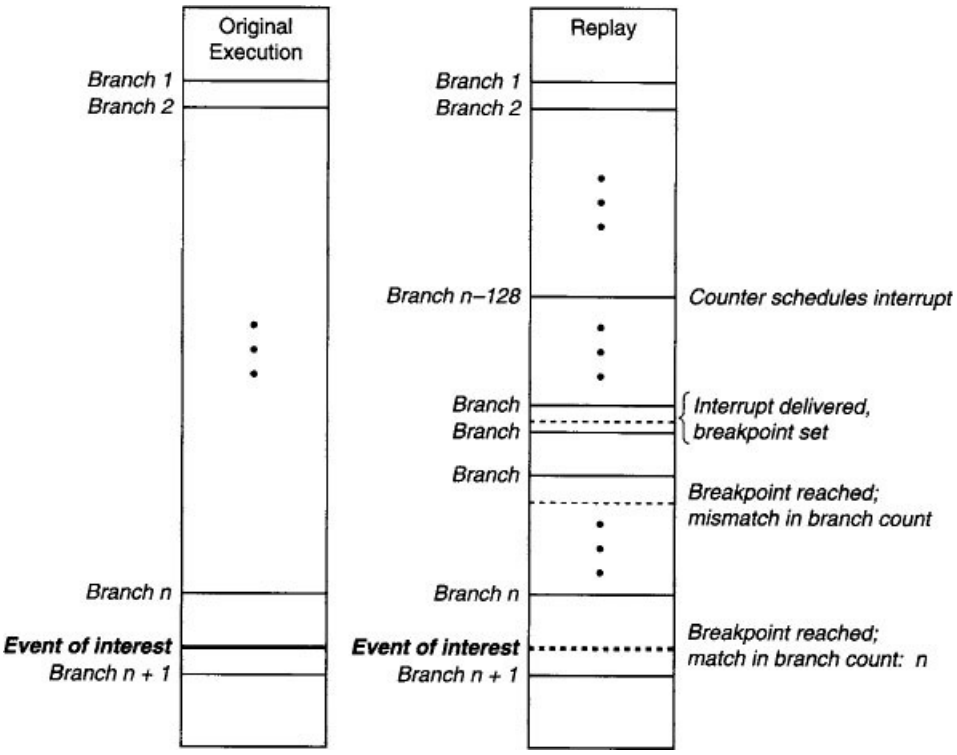


Figure 10.5 Two-Phase Method for Delivering a Nondeterministic Event Accurately During Replay.

The number of events to be recorded can grow to an unmanageable size unless one takes care to minimize the logged information. Note that only those events that can affect the execution of the monitored virtual machine need be logged; it is not necessary to log the events that arrive at the VMM but are not delivered to the virtual machine itself. Some of the common events that must be logged are timer and I/O interrupts. There are some cases where, in addition to logging the event itself, it is necessary to log the actual input value, for example, for inputs from the keyboard, mouse, network interface card, real time clock, and CD drive. Input from the hard drive need not be recorded if we ensure that the hard drive will be set to the original state at the start of the replay. Even in the case of a CD drive, the actual input values read from the media need not be recorded if the original CD is available to be replayed; in this case, even these events become deterministic.

It is important to have good knowledge of the characteristics of the specific system to ensure that all nondeterministic events are properly logged. As an example of system-specific aspects that need to be considered, the ReVirt paper (**Dunlap et al. 2002**) indicates the following three aspects specific to the IA-32.

1. The IA-32 architecture allows a long memory instruction (e.g., a string instruction) to be interruptible, meaning that an asynchronous interrupt could cause a context switch in the middle of execution of the instruction. By saving the value of a register, ecx, as part of the state, an implementation will have the necessary information to resume the instruction where it left off after handling the interrupt. This implies that the register value must also be saved when a nondeterministic event is recorded, in case the interrupted instruction happens to be a string instruction.
2. The read-timestamp-counter (rdtsc) instruction returns nondeterministic values, but this instruction does not trap when in user state unless the control register CR4 is set in a particular way. One way to deal with this instruction is for the VMM to treat it the same way as other sensitive but nonprivileged instructions, as described in **Chapter 8**, namely, to scan ahead for the instruction, insert a trap, and emulate the instruction. The ReVirt system sets the CR4 so that the

trap, and emulate the instruction. The ReVirt system sets the CR4 so that the instruction traps, replaces the instruction with a library call routine **gettimeofday(),** which does not use the rdtsc instruction, and scales the value returned to account for the overhead.

3. A similar situation occurs with the read-performance-monitor-counter (rdpmc) instruction. This instruction also exhibits nondeterministic behavior and usually does not trap in user mode as in the rdtsc case. As with the rdtsc, the ReVirt system sets the CR4 so that the instruction traps in user mode, but then it simply disallows the instruction in the guest kernel and applications running in the monitored virtual machine.

The ReVirt system was built on a hosted virtual machine system based on Linux (UMlinux 2002). The proper functioning of the system—specifically the ability to detect the occurrence of an external interrupt event as well as to deliver it at the precise point during the computation—was tested using special kernels. The performance overhead due to logging was found to be less than 8% for a wide variety of benchmarks. The growth rate of the log was found to be between 40 and 1400MB per day. This is a reasonable number and, in practice, allows logging for several days, possibly even months.

10.1.4 Role of Dynamic Binary Rewriting Technology in Security

In **Section 4.7** we described a type of process virtual machine for which both the host and guest ISAs are identical but where the principal aim is the optimization of code for a particular platform. In such a system, a program is executed under the control of runtime software, which interprets the instructions in the program, determines code regions (e.g., superblocks) that are frequently executed, translates such code regions, optimizes them, and saves the optimized binary in a code cache from which the code is executed when encountered next.

The ability to control the execution of a program through the runtime also makes it possible for the runtime to sandbox the execution of a program, both to prevent the program from being attacked by

potential intruders and to prevent the program from being the launching point of an attack on the system. This is the principle behind *program shepherding* (**Kiriansky, Bruening, and Amarasinghe 2002**), a concept that has been implemented on the RIO dynamic optimization infrastructure (**Bruening, Duesterwald, and Amarasinghe 2001**), which itself is an evolution of the Dynamo binary optimization system (**Bala, Duesterwald, and Banerjia 2000**) described in **Chapter 4**.

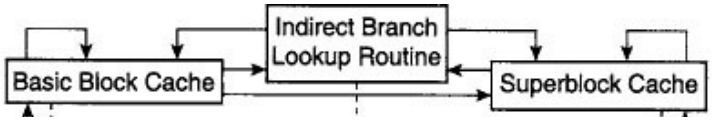
The fundamental idea behind program shepherding is that a program can be adequately protected by ensuring that every branch or jump is to a legal location and every branch into a code region originates from either another part of the same program or from some other trusted location on the system. This is not done easily at compile time because static analysis fails to resolve addresses of dynamically shared libraries as well as the targets of many indirect jumps. Thus safe execution of a program can be ensured only through runtime checks.

Restricting Control Transfers

Let us now examine the way in which control transfers can be restricted. One way to do this is to add instrumentation (i.e., instructions) at branch and jump sites to check the validity of the target. However, this could degrade performance of the program quite significantly. Moreover, as long as the program is vulnerable to attack, the attacker could change this instrumentation to circumvent the checks. A more effective way of achieving this is through techniques used in dynamic binary optimization. A section of code is interpreted the first time it is executed. The targets of the conditional and indirect branches are recorded in a safe location. When the same sequence of basic blocks has been executed a predetermined number of times, this sequence of blocks is gathered into a superblock, reoptimized, and placed in a code cache, as described in **Chapter 3**. The runtime software of such a system performs checks for the validity

of branch targets both during the interpretation phase as well as during execution from the code cache. If the superblock represents the typical path of execution, these checks will be performed much less frequently when executing out of the code cache. In order to ensure that the code being executed is not maliciously modified, the code cache itself should be protected by making it writable only when the runtime is in control.

An overview of the RIO system infrastructure is shown in **Figure 10.6**. The specific characteristic of the system, namely, that the RIO code and the application code both work as part of the same process and share an address space, is an inheritance from the Dynamo system, which was described in **Section 4.7.2**. Even though they are part of the same process, there are two modes in which the machine operates, the *RIO mode,* in which the infrastructure code operates, and the *application mode,* in which the application work is performed. The system begins by scanning instructions and collecting basic blocks, which are inserted into a *basic block cache,* a region of memory protected from being overwritten. At a branch or jump, a map (hash) table lookup is performed to check whether the next basic block has already been cached. If it has been cached and if a direct address is given, the two basic blocks in the cache are linked through a new direct branch so that the table lookup is avoided the next time. Further performance improvement is obtained by stitching together frequently executed sequences of basic blocks to form a superblock, which is then inserted into the code cache (shown as the *superblock cache* in **Figure 10.6**). The collection of basic blocks and superblocks is done in RIO mode, but the execution of the code itself is done in application mode. A mode switch into the RIO mode occurs when a control transfer lookup fails to find an entry either in the basic block cache or in the code cache.



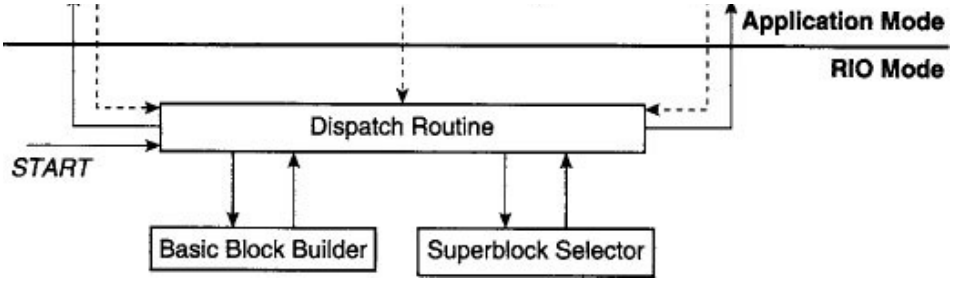


Figure 10.6 Flowchart of the RIO System (**Kiriansky, Bruening, and Amarasinghe 2002**). *Dotted arrows depict performance-critical functions, with the operation switching between application mode and RIO mode.*

Management of control flow transfers is achieved as follows. For each direct branch from one cached basic block to another, security checks are performed at the time the basic blocks are linked. A link is added only if the security policy allows a direct transfer; otherwise control returns to a runtime routine that checks for and handles all potential security violations. Once the link is inserted, it is part of the basic block cache (or superblock code cache), which itself is protected from being tampered with. As long as the program is restricted to such linked branches, there is no overhead in execution. In fact there could even be a performance benefit due to more efficient layout of the code.

Indirect-jump destinations have to be looked up in the map table to determine the mapping of destination addresses to corresponding locations in the basic block cache or the code cache. Only those cache addresses that are validated are placed in the map table; these account for most of the addresses to which jumps are made. Additional checks can be recorded in map table entries. For example, the target of a return from a subroutine can be checked to make sure the program returns to a point immediately after the original call. Such a check would thwart the type of attack that was described in **Section 10.1**, where the program returns to an inappropriate address because the stack has been clobbered.

In general, the only branch or jump targets that lead to any significant program degradation are indirect jumps targeted outside the

code segment being protected. The RIO system checks for the validity of such targets by performing dynamic checks whenever they are encountered.

Restricting Code Execution

The origin of a piece of code must be authenticated before it is allowed to execute. This authentication must be performed not only at the time the code is loaded from the disk or from the network, but also while it is executing—the code could be changed in memory either by another program on the system or even by itself. Any changes to the code in memory must either conform to the security rules or be disallowed.

Most of the checks in the RIO system of this kind are performed at the time the system copies a sequence of instructions into the basic block cache. The main challenge here is to ensure that the code was not tampered with after being fetched into memory and before being copied into the basic block cache. Modern program binary file formats separate the code sections from the data sections and allow loaders to write-protect code pages as they are brought into memory. Any attempt to change the code results in an exception. However, the system also needs to handle the case of dynamically generated code, permitting the execution of such code when it is legal to do so and preventing it when it is not.

There are two cases of importance. The first is the case of new sections of code that are created and then loaded for execution. In this case, the region of code is usually converted from being writable to being write-protected. The RIO system authenticates the code at this point and allows the conversion only if the request is from trusted code. Similarly, if the program makes a direct jump into writable space, the system verifies the origin of the jump and write-protects the target area before allowing execution to proceed.

The second case occurs when code and data are shared on the same page—this case poses an interesting problem. Since data areas should not be write-protected, it is necessary to have two copies of such pages, one of which is write-protected and is the source of instructions, while the other serves as the target for data reads and writes and is not write-protected. This solution is adequate for most common situations. It does not handle the situation where the code itself is changed by the program, however—the case of self-modifying code. Such writes into the code area must be intercepted by the runtime monitor in order to ensure they are being performed legally. It is necessary for the system to monitor all writes into write-protected regions and to authenticate the source of such writes. If it is a legal self-modifying code situation, the runtime monitor makes the page writable, allows the write to proceed, and then write-protects the code page again, after ensuring that the basic block cache and the code cache are purged of all the affected regions. This is clearly a case that could significantly degrade the performance of the system. Fortunately, it is not a common occurrence in modern programs.

TABLE 10.1 Page Access Privileges in the RIO System

Page Type	RIO Mode	Application Mode
Application Code	R	R
Application Data	RW	RW
RIO Code Cache	RW	R (E)
RIO Code	R (E)	R
RIO Data	RW	R

R — read only, RW — read/write, R (E) — executable

Protecting the Runtime Monitor

The RIO system designers protected the RIO system code itself from attack by using different page protections during the RIO mode of operation and the application mode. The RIO mode has slightly higher page access privileges than the application mode, even

higher page access privileges than the application mode, even though both modes operate within the same address space. An emulated "context switch" occurs when the machine switches modes. **Table 10.1** indicates the access privileges to the various types of pages in the RIO mode and in the application mode. For instance, the code cache itself has read and write privileges in the RIO mode, whereas it has only read and execute privileges in the application mode. Tables used by the RIO system, for example, the map tables, have read-only privilege in the application mode whereas they have read and write privileges in the RIO mode.

10.1.5 Secure Systems of the Future

Security is an aspect of computing that has not been systematically addressed in the past, and is beginning to gain tremendous importance. Users and organizations are already beginning to think of ways in which they can partition their platforms so that they are able to run open, general-purpose, widely available systems and applications that are insecure alongside other applications and data that have to be protected and kept highly secure. The Terra system (**Garfinkel et al. 2003**) attempts to achieve this by creating a *trusted virtual machine monitor* that supports virtual machines incorporating the semantics of dedicated, tamper-resistant hardware on the same hardware platform as virtual machines that run normal, open, general-purpose applications.

We have seen in this section the application of various techniques described in earlier chapters of this book to handle the security problem. The process of thinking through the various issues and of coming up with solutions to handle them has motivated and will continue to motivate the search for fundamental ways of avoiding thorny security issues. Thus the world is moving toward new paradigms where security, perhaps even at the expense of some performance, is a fundamental aspect in the design of an environment and its applications. The HLL VMs, as described in **Chapters 5** and **6**, ex-

emplify this approach. For some time to come, however, the vast legacy of existing environments will continue to call for security solutions of the type discussed in this section.

10.2 Migration of Computing Environments

It is not uncommon for individuals to maintain programs and to work on multiple computers at different locations, for example, one at home and one at work. Usually, there is an overlap in the activity conducted in the user's different work locations, and it is clear that the user's data is likely to be needed at multiple locations. To facilitate this multiple work-site environment, it is likely that the same operating system and the same set of applications are installed on different computers. Yet there are often subtle differences between the environments of the various installations that prove frustrating to the user. For example, one location may have a slightly different version of an application program from another, perhaps because an upgrade was applied to the machine at one location and not the other. One might naturally ask whether there is some way in which the user can see an identical environment everywhere he or she works.

Many users get around this problem by working exclusively on portable devices such as laptop computers. While this ensures that the user always sees and works with exactly the same environment, it does have two significant disadvantages. First, the user is required to carry a physical device to transport the environment, and second, this means that the user is responsible for the physical security of the portable device at all times.

In the 1960s, mainframe servers solved this problem (at least within the environs of a large office building) by providing users with simple, "dumb" terminals as the only interface to the system. The oper-

ple, "dumb" terminals as the only interface to the system. The operating system, applications, and data of the user resided in a central location and were accessible from the terminal through a network of serial links. The system virtual machine environment described in **Chapter 8** extended this concept by allowing each user to perceive the computer being accessed as his or hers exclusively. The virtual machine concept provides flexibility, by allowing each user to customize his or her environment, and security, by isolating a particular user's environment from those of other users of the machine.

The last few decades have seen a dramatic decrease both in the cost and in the size of computing devices. The dumb terminal of the mainframes has now become a sophisticated computer in its own right. Thus, while the concept of the mainframe server still survives, most users today access both mainframes and the Internet through a device capable of performing significantly sophisticated computation. Such processing may include applications such as graphics rendering and game simulation, Web browsing, and the editing of text documents. It is the environment associated with this interface that needs to be replicated when a user moves from place to place.

Replicating such an environment from one computer to another is not trivial, because this implies that a tremendous amount of state information must be transferred. The state of the machine includes not only the state of the resources used by the operating system and the applications running on the machine, but also the code and data belonging to the operating system and applications. Thus, a *capsule* is formed to capture the state of a running machine and hence information about the processes currently active on the system. The notion of a capsule allows a user environment targeted for migration to be treated as an object that can be manipulated.

Even assuming that the actual hardware on multiple sites is identical, replicating the environment requires all this state information

to be encapsulated, transmitted across a network, and installed on the other computer before the user can take over operation at a second computer. These operations have to be done efficiently in order for the switch between environments to appear seamless to the user. The following subsections will be devoted to ways to make such environment mobility practical and efficient.

10.2.1 Virtual Computers

As we mentioned in the introduction, virtual machine technology provides the ability to capture the entire state of a computer system and thus facilitates migration of a full computing environment. **Figure 10.7** shows what is involved in such a migration. Traditionally, in order to migrate from one computer to another, one would close all programs on one computer, shut it down, and either transport the hard drive or copy all information contained on the hard drive of the first computer on to the hard drive of the second.

Often even this seemingly simple scenario runs into complications because of subtle differences in the configurations of the two machines. Virtual machine technology inherently is better at smoothing out such subtle differences. The process of environment encapsulation is simplified if the environment is running on a virtual machine rather than directly on real hardware. The virtual machine monitor must implement data structures that separate the resources belonging to the different virtual machines running on the system. Thus, the virtual machine itself can serve as a capsule that can be migrated to another system that implements the same virtual machine system, as shown in **Figure 10.7**b.



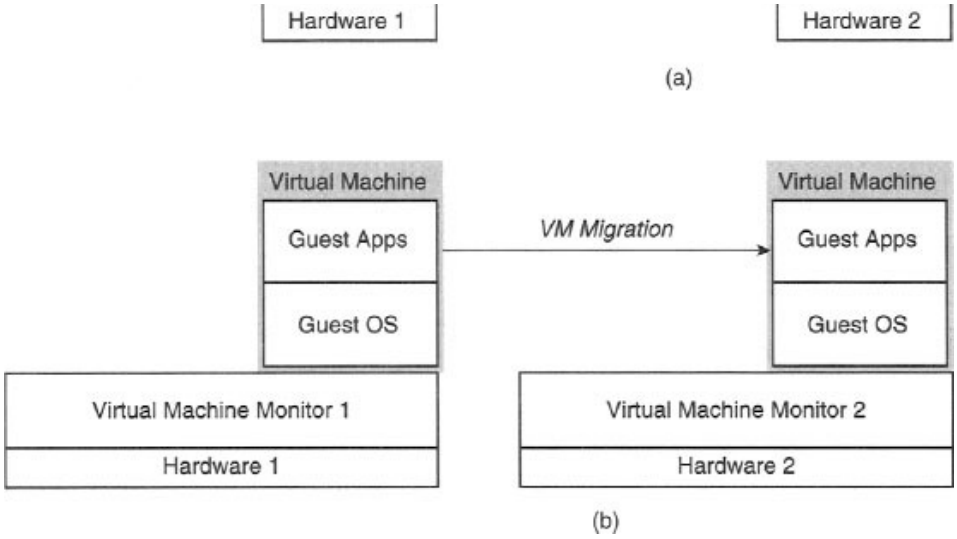


Figure 10.7 Environment Migration Using Virtual Machines. *(a) Traditionally, migration involves the movement of data between machines, each with its own hardware, operating system, and complement of applications. (b) A faithful reproduction of the user environment is possible from one machine to another through the use of virtual machine technology. The state of the virtual machine, including the state of the application, and the state of the operating system are packaged and migrated. The data for the application may also be migrated if it is not located on a file system accessible at the destination.*

The information about a virtual machine that is typically kept for use by the virtual machine monitor is restricted to the configuration of the virtual machine and pointers to tables that link virtual resources such as memory to real resources on the system. A capsule, on the other hand, does not contain any information about the mapping of the virtual resources. A capsule must also capture the state contained in all resources, including the disks and memory. It is conceivable that the information contained in a capsule can be many gigabytes in size and could take several hours to be transmitted on networks commonly used for connecting home computers to the outside world, for example.

Encapsulation of a running machine is similar to checkpointing. It allows the machine to be suspended for an indefinitely long time and resumed exactly from the point where the execution was originally suspended. The capsule is a host-independent checkpoint that can be transported from one machine to another; execution can be suspended on one computer and resumed at exactly the same point on another computer. If the virtual machine architecture is identical

to some real machine architecture, the execution can be resumed on a real implementation rather than on another virtual machine system.

It must be remembered that the system virtual machines running on a centralized mainframe such as the IBM z/VM also allow a user to see the same environment when traveling from one location to another. However, in that case, both the data and the processor reside in a remote location and stay there, even during execution. The amount of state information that needs to be transferred to a remote terminal is negligible, and the overhead of updating the environment to make it cognizant of the new terminal is small. The harder problem is the one we have outlined here—that of migrating the environment to work on different data and compute facilities.

There are several issues involved in developing a practical solution for this problem, and we list some of them here.

- The entire state of a computer (which could include tens of gigabytes or more of disk space) can be so large that the time taken to migrate the state may be prohibitive. Note that it is generally not necessary to send the entire state from one computer to another initially. It is usually sufficient to transmit a small part of the state at first, transmitting additional parts of the state as needed. It may also be possible, as we shall see later, to take advantage of information that may already be available on the second computer, e.g., portions of the operating system itself.
- Having decided what portion of the machine state needs to be migrated, there is still the problem of packaging and securely transmitting the information. Compression and encryption techniques have to be employed to reduce the amount of data transmitted and to transmit it securely.
- When the hardware on the two machines is identical and when the virtual machine monitors running on them are identical, the process of transmitting the environment from one machine to another should be seamless. It may be the case that while the processors are identical in the two machines, the memory or I/O configuration on the two machines is different. For example, one machine may be connected to the external network using an Ethernet adapter, the other with a wireless connection. We have already seen in **Section 8.4** how such situations are handled, both in native system virtual machines and in hosted virtual