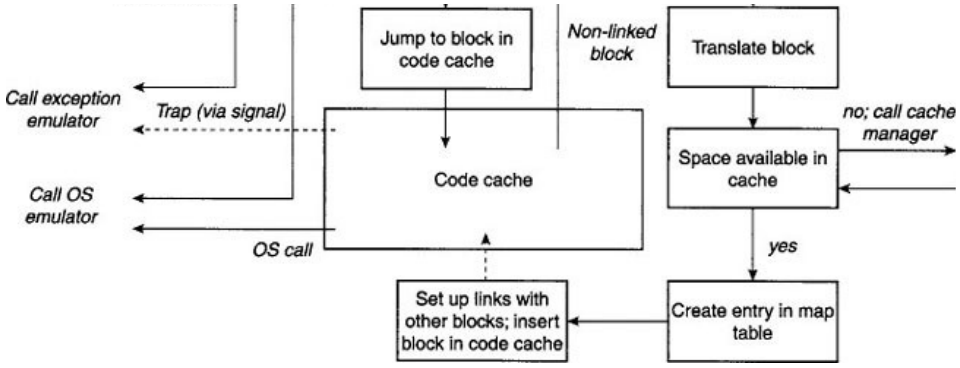


Figure 3.19 Execution Flow of an Emulation Engine.

The staged emulation strategy just described begins with interpretation and moves to binary translation of basic blocks, as described in the previous chapter. A common additional optimization is to combine basic blocks into larger units for translation and optimization. A frequently used translation unit of this type is the *superblock,* a block of code with a single entry point but possibly several exit points. Superblocks and superblock optimization are discussed in **Section 4.3**. The techniques discussed in this chapter work with basic blocks as well as with larger translation units, such as superblocks. The three levels of emulation—interpretation, binary translation of basic blocks, and binary translation with optimization (on superblocks)—allow a number of staged interpretation strategies. For example, here are two commonly used strategies.

- Interpret with simple profiling. Then when a threshold is reached, generate and optimize superblocks that are held in the code cache. This method is used in the FX!32 system (**Hookway and Herdeg 1997**), the Aries System (**Zheng and Thompson 2000**) and the HP Dynamo same-ISA optimizer (**Bala, Duesterwald, and Banerjia 2000**).
- Skip the interpretation stage and immediately perform simple binary translation on dynamic basic blocks; the translation includes code for gathering profiling data on the basic blocks. As translated basic blocks become hot, form superblocks and optimize the superblocks. This method is used in the Sun Wabi system (**Hohensee, Myszewski, and Reese 1996**), the IA-32-EL system (**Baraz et al. 2003**), and the Mojo same-ISA optimizer (W.-**K. Chen et al. 2000**).

3.6 Exception Emulation

Exception Handling

Exception conditions can occur at virtually any time during instruction execution, often unpredictably. The correct handling of exceptions poses some of the more difficult challenges for compatible emulation. We use the term *exception* to denote any trap or interrupt. A *trap* is the direct result of program execution and is produced by a specific instruction. An *interrupt* is an external event, for example, caused by the I/O system, and is not associated with a particular instruction.

We use the conventional definition of exception preciseness—i.e., an exception is *precise* if (1) all instructions prior to the faulting instruction have been executed, (2) none of the instructions following the faulting instruction have been executed, and (3) without loss of generality, the faulting instruction has not been executed. (If the ISA semantics say that it should be executed, then this adjustment can easily be made.) We assume that all exceptions in both the source and target ISAs are defined to be precise. In the case of an external interrupt, we define preciseness with respect to some instruction being executed at approximately the same time as when the interrupt occurs.

For a process VM implementation, we further divide exceptions into the following two additional categories.

ABI visible—These are exceptions that are visible at the ABI level. They include all exceptions that are returned to the application via an OS signal; i.e., they are a function of the source operating system as well as of the user-level ISA. For example, if the guest OS allows a signal for a memory protection fault, then the trap becomes ABI visible. This category also includes those exceptions that cause the application to terminate (as defined by the guest OS) because at the point of termination a precise application state may be required.

ABI invisible—There is no signal and the application does not terminate when the exception occurs because the ABI is essentially unaware of its existence. An example of an invisible exception is a timer interrupt that the OS uses for scheduling. Depending on the signals supported by the host OS, page faults may also fall in the category of ABI-invisible exceptions.

3.6.1 Exception Detection

Traps, caused as a by-product of instruction execution, are the more difficult type of exception to deal with, and we discuss them first. During instruction emulation, traps can be detected in one of two ways. First, a trap condition can be explicitly checked as part of an instruction's interpretation routine. For example, overflow caused by an add instruction can be detected by explicitly comparing the input operands and the final sum. We refer to this as *interpretive trap detection.* If the interpreter routine finds a trap condition, it jumps to a runtime trap handler. Second, a trap condition can be detected by host platform hardware when an instruction in the emulation code is executed. That is, a target instruction traps because the corresponding source instruction being emulated would also have trapped. If there is a signal supported by the host OS that matches the trap condition, then the signal can be delivered to the runtime trap handler. Clearly, ignoring efficiency issues, the first method, checking via interpretation, can always be done. On the other hand, the more efficient second method is dependent on the semantic match between source and target ISAs, details of the emulation process, and support from the host OS.

In order for the host platform to support detection of exceptions as a by-product of emulation, a key element is the host OS signal mechanism. To implement this method, the runtime should register all exceptions for which the host OS supports signals at the time the runtime is initialized. However, as emulation proceeds, if the guest application happens to register a signal by performing a system call,

the runtime OS emulation code intercepts the system call and records it as a “guest-registered” signal in a table.

During emulation, when an exception occurs that causes a trap to the host OS, the host OS delivers the appropriate signal to the runtime. At that time, the runtime checks the guest’s signal table to see if the guest had registered the signal. If so, the runtime adjusts the guest state so that it appears that the guest’s OS is signaling, and it transfers code to the guest’s signal-handling code. Otherwise, the runtime handles the trap condition indicated by the signal.

There are three cases to be considered, depending on the nature of the trapping condition.

1. The trapping condition is also ABI visible on the host platform. In this case, the runtime signal handler will be invoked, and it can take the proper action as just described.
2. The trapping condition is not ABI visible in the host platform. In this case, the earlier approach will not work and explicit checks for the trapping condition will have to be placed in the emulation code along with a jump to the runtime if the trap condition holds; i.e., interpretive trap detection must be used.
3. A trapping condition is visible in the host ABI but not in the guest ABI. In this case, there may be extraneous traps. Here, the runtime trap handler will have to determine whether the trap condition would have been visible to the source instruction.

In some cases, multiple guest trap types may map to one host trap type. For example, the host may have a single exception condition that covers both integer and floating-point overflow, whereas the guest may have separate exceptions for the two types of overflow. In this case, the runtime must inspect the trapping instruction to determine the exception to be reported to the guest.

3.6.2 Interrupt Handling

Some interrupts may be ABI visible; that is, the application may register a signal handler for certain interrupt conditions. When inter-

rupts occur, they do not necessarily have to be handled immediately because they are not associated with any particular instruction. In general, there is an acceptable response latency during which the emulated application may be put into a precise state before control is transferred to the interrupt handler. Typically, at least several tens or hundreds of instructions can be emulated within the acceptable response latency.

Because the runtime has registered all signals, if an interrupt does occur, the signal will first be delivered to the runtime. If interpretation is being performed, the current interpretation routine can be completed before the runtime passes control to the guest interrupt handler. A realistic interrupt response latency is usually long enough to accommodate the completion of an interpreter routine.

If binary translation is being used for emulation, however, the situation is more complicated. When an interrupt occurs and a signal is delivered to the runtime, execution within a block of translated code may not be at an interruptible point. Generally, when code is translated, side tables are maintained so that if a source instruction traps, it is possible for the runtime to provide a correct precise state for the trapping instruction (these mechanisms are described later). This is not the case with interrupts, because essentially every instruction is interruptible and the constraints for precise state recovery at arbitrary points in the translated code would be too restrictive.

A further complication of binary translation is that when the runtime passes control to chained translation blocks, it may be an arbitrarily long time before control gets passed back to the runtime. If, for example, execution is within a loop of several translation blocks chained together, a very long time may elapse before the loop is exited. This means that special steps may have to be taken to provide an acceptable interrupt response time. To summarize the problem, when an interrupt signal is delivered to the runtime, the translated

code may not be in an interruptible state. Yet if the runtime passes control back to the translated code and "waits" until the code returns to the runtime (at which time it is in an interruptible state), the runtime may not reassume control within a required interrupt response time window.

To solve the interrupt response problem, the following steps can be followed. (1) When the interrupt occurs, a signal is sent to the runtime and control is transferred from the currently executing translated code block to the runtime. (2) After receiving the signal, the runtime eliminates the possibility of a chained loop within the translated code blocks by locating the currently executing translated block and unlinking it from subsequent translation blocks, e.g., by overwriting the link at the end of the current translated code block so that it jumps back to the runtime when it is finished. (3) The runtime returns control to the translated code at the point where the interrupt signal was received. (4) The current translation block finishes and then jumps back to the runtime. (5) The runtime handles the interrupt. Note that this method assumes translation blocks have two properties: (a) There are no loops internal to a single translation block; (b) at the end of each translation block, the target code is at a precise interruptible state. Both of these assumptions are typically provided in binary translation systems, and neither unduly burdens the translator or inhibits performance.

3.6.3 Determining Precise Guest State

After an exception condition has been discovered, the runtime must be able to provide the proper precise state for the emulated guest process. If interpretation is being used, this is fairly straightforward; but if binary translation is used, the process of determining the precise state may be much more complex, especially if the translated binary instructions have been reordered or otherwise optimized. The following subsections discuss precise state restoration during both interpretation and simple binary translation; the difficulties

both interpretation and simple binary translation, the difficulties brought about by code reordering and optimization are discussed in the next chapter.

Interpretation

With interpretation, instructions are typically emulated one at a time in the original program sequence. Consequently, the source PC is updated as interpretation proceeds. Furthermore, the correct source state (both memory and registers) is updated at each instruction boundary. For example, consider the Add interpreter code given earlier in **Figure 2.4** repeated here in **Figure 3.20**). If the add overflows, it may be detected via a trap when the sum = source1 + source2 statement is executed. At this point in the interpreter routine, the source PC has not yet been updated and still points to the (overflowing) add integer instruction. The result register RT has not been updated (it can be updated via the runtime signal handler if specified by trap semantics).

The signal handler in the runtime will be provided with a *target* PC that points to the sum statement in the interpreter routine. However, the runtime signal handler should use the *source* PC (maintained during interpretation) to go back into the original source code to find the source instruction that was being emulated at the time the trap occurred. Note that using the source PC maintains portability of the interpreter code; i.e., its behavior is not tied to specific locations of interpreter code.

```
Add:
        RT = extract(inst,25,5);
        RA = extract(inst,20,5);
        RB = extract(inst,15,5);
        source1 = regs[RA];
        source2 = regs[RB];
        sum = source1 + source2;
        regs[RT] = sum;
        PC = PC + 4;
        If (halt || interrupt) goto exit;
        inst = code[PC];
        opcode = extract(inst,31,6);
```

```
extended_opcode = extract(inst,10,10);
routine = dispatch[opcode, extended_opcode];
goto *routine;
```

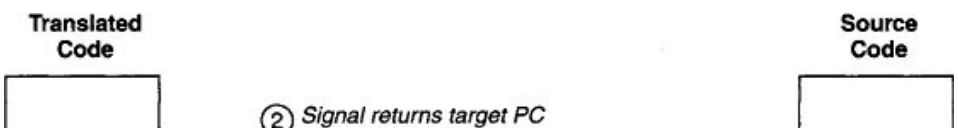
Figure 3.20 Add Interpreter Routine. *Overflow during the add operation may trap and cause a signal to be delivered to VM software. The source PC will point to the trapping source instruction.*

Binary Translation: Locating the Program Counter

With binary translation, the restoration of state following an exception begins by determining the source PC of the faulting instruction. Then given the PC, the rest of the precise application state is recovered. In this subsection we focus on determining the source PC of the trapping instruction.

One difficulty in determining the PC of the trapping source instruction is that at the time the trap occurs, the runtime is provided with the PC of the translated target code rather than the source PC. Unlike interpretation, however, binary translation typically does not keep a continuously updated version of the source PC. This means that there must be some indirect mechanism for finding the source PC of a faulting instruction, given the translated target PC. In order to track down the corresponding source instruction, it is useful to keep PC-related information in a side table.

For recovering the precise source PC, a *reverse translation* side table may be used (see **Figure 3.21**). The most straightforward implementation of a reverse translation table contains <target PC, source PC> pairs, indicating for each target PC (i.e., an address in the translated binary) the corresponding source PC from which it was translated. Then given the PC of a trapping target instruction, the runtime can scan the table for the trapping instruction's PC; when it finds a match, the corresponding source PC, the second half of the pair, becomes available.



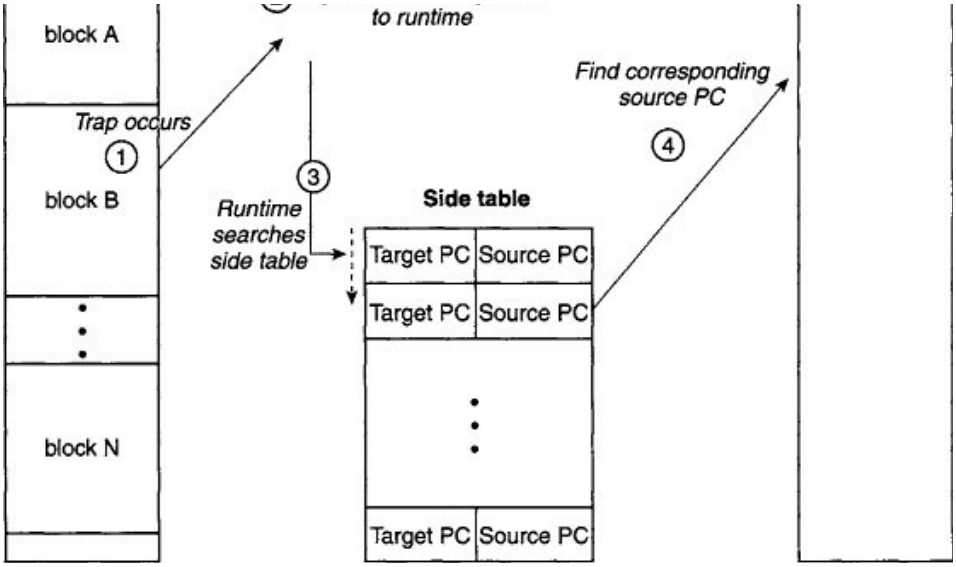


Figure 3.21 Finding the Trapping Source PC, Given the Target PC. *(1) The trap occurs and (2) the signal handler returns the target PC to the runtime software. The runtime (3) does a search of the side table (4) to find the corresponding source PC that caused the trap.*

The table, in the form just described, has a couple of inefficiencies. First, the target PC lookup may require a linear scan of the table. Second, the table will be fairly large; because it holds a pair of address pointers for each translated instruction, the table could easily be larger than the translated code. The linear scan can be replaced with a binary search if the target PCs are in sequential numerical order. For some code cache management algorithms, e.g., those using FIFO replacement, this will be the case naturally. And, as we will see in **Section 3.8.2**, FIFO replacement is one of the better strategies for code caches.

The table size can be reduced by taking advantage of locality in target and source PC values. For example, one could hold a subset of the PCs to their full length, with others being expressed as deltas from the full-length versions. The PCs at the beginning of each translation block can be kept in full, while PCs within the translation block are deltas from the initial PCs. A further simplification can be made if the target ISA has fixed-length instructions. In this case, the side table needs to contain only an array of source PCs that can be accessed directly via the target PC values.

An additional complication occurs if a given target instruction corresponds to more than one source instruction (so the target PC corresponds to more than one source PC). This can happen, for example, when a RISC ISA is mapped to a CISC ISA. Two RISC instructions, a load instruction and an ALU instruction, may be translated to a single CISC instruction that both loads from memory and performs an ALU operation. In this case, when a trap occurs, it may be difficult to identify which source instruction is the one with the exception (although in the example just given this is not the case). A related complication occurs when the translated code is rearranged or optimized so that the target instructions are executed in a different program order than are the corresponding source instructions. Both of these issues will be discussed in greater detail in **Chapter 4**; generally speaking, the solution is to identify the beginning of the translated block that contains the trapping source instruction and then to return control to the runtime, which can analyze and/or interpret the original source code to sort out the correct source state and PC value.

To support the required runtime analysis (and to reduce side table space requirements), the side table can be organized according to translation blocks, with an entry for each block. This entry holds the beginning target PC along with enough information to allow reconstruction of the complete translated code block. Refer to **Figure 3.22**. If the translation block is contiguous, this information would be no more than a count of the number of translated source instructions. If the block is noncontiguous, then a sequence of start source PCs and counts would suffice or, even simpler, the initial source PC and control flow information used when forming the initial translation, e.g., the taken branches and/or jump targets. Given this information, the runtime interprets the source code to identify the trapping source instruction. To optimize the process, a hybrid side table structure can be used. For example, if the same target instruction frequently traps, then its corresponding source PC can be cached in

a side table to avoid repeated analysis and interpretation.

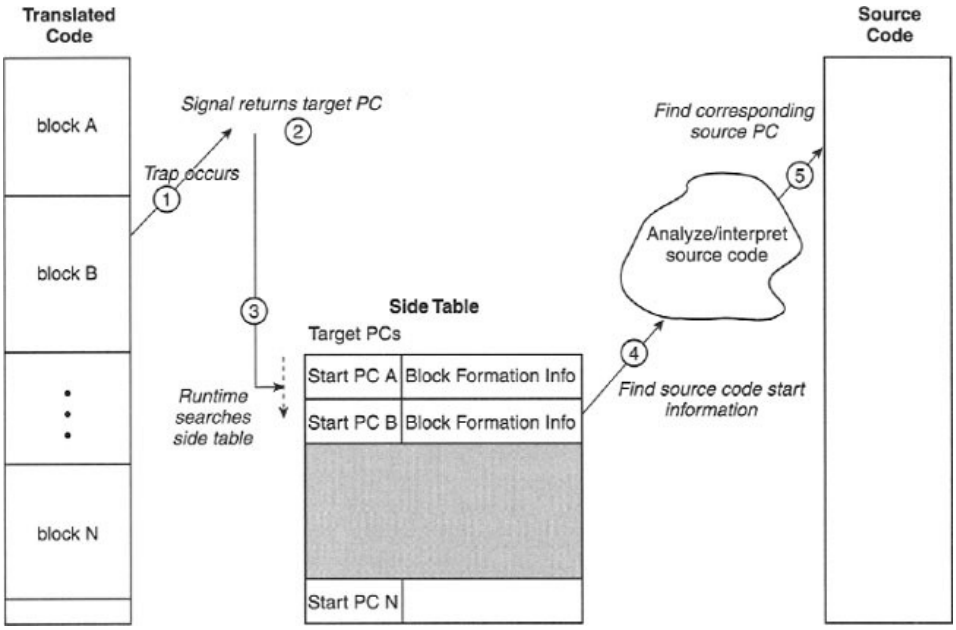


Figure 3.22 Using an Optimized Side Table to Find the Trapping Source PC. *(1) The trap occurs and (2) a signal delivers the target PC to the runtime. (3) The runtime then performs a search to find the trapping translation block. (4) The table entry contains enough information to allow analysis and/or interpretation of the source code blocks. (5) To find the source PC corresponding to the trapping target PC.*

Binary Translation: Register State

When an exception occurs during execution of translated code, the correct, precise register state must be restored, just as it would be in the original source code. In this subsection, we consider methods that can be used when the optimization level of a translated binary is relatively low. In particular, it is assumed that no code reordering is done and no register state updating instructions have been removed due to optimizations. Other, more complex register restoration methods are often closely tied to the specific optimization techniques used, and the discussion of these methods is deferred until **Chapter 4**, where binary optimization is covered in depth.

The simplest case occurs when the emulator uses a consistent source to target register mapping and register state is updated in the same order in both source and translated code sequences. By a *consistent* register mapping, we mean that the mapped location of a

sistent register mapping, we mean that the mapped location of a specific source register (either in a target register or in memory) remains the same throughout emulation. In this case, the source register state can be recovered from the target register state at any point.

A slightly more complex case occurs if the source-to-target register mapping varies between or even within the translation blocks, but the order of register updates is the same in both source and target code. In this case, a side table for the translated code blocks can be used to indicate how the register mapping should be restored. Alternatively, as in the case of trap PC identification, the source code can be analyzed again from the beginning of a translation block, the current register assignments regenerated, and the resulting information used to restore the correct source register state.

Binary Translation: Memory State

Memory state is changed by store instructions. As long as the source program store instructions are all emulated in the original program order, it is fairly straightforward to maintain precise memory state. If, in addition, all potentially trapping source instructions are emulated in the same order with respect to the store instructions, then the memory state at the time of a trap can be guaranteed to be consistent with the source PC that is recovered.

If code is reordered, then, as we shall see in **Chapter 4**, the presence of memory stores constrains the reordering that may be done. In particular, a potentially trapping instruction cannot be moved below a store that follows it in the original source code. Otherwise, because memory has been overwritten, the memory state will not be recoverable if the moved instruction does trap. Of course, there are optimizations (or hardware) that can buffer memory stores, but we defer discussion of these techniques to later chapters.

On the other hand, if a trapping instruction is moved ahead of a store, then it may be possible for the runtime trap-handling code to

store, then it may be possible for the runtime trap handling code to emulate (typically via interpretation) the store instruction to complete the store update. Again, because it is tied in with code-optimization methods, details of this technique are deferred to **Chapter 4**.

3.7 Operating System Emulation

The operating system interface is a key part of the ABI specification, just as the user-level instructions are. However, because a process-level VM is required only to maintain compatibility at the ABI level, it does not emulate the individual instructions in the guest's OS code; rather it emulates the function or semantics of the guest's OS calls, typically by converting them to host OS operations. There are two important cases to be considered, one of which is easier to implement than the other (although neither is easy!). In the first case, the guest and host operating systems are the same; e.g., they are both Linux or Windows. The other, more difficult case occurs when the operating systems are different. In this section, we discuss both cases, beginning with the simpler, "same OS" case.

3.7.1 Same-Operating-System Emulation

A runtime emulates a guest's OS by a combination of host OS calls and operations performed by the runtime itself. The basic emulation techniques are discussed briefly in the following subsections.

Operating System Call Translation

When the guest and host operating systems are the same, the problem of OS call emulation is primarily one of matching the OS interface syntax. The OS functions required by the guest are available in the host, but it may be necessary to move and format arguments and return values, possibly forming some data conversions in the process. For example, an OS running on a platform with relatively

few registers, such as the IA-32, may pass arguments in a memory resident stack, while the same OS running on a RISC-based platform with many registers may pass arguments in registers. In this case, arguments must be copied from the stack to registers, or vice versa, when emulating a system call. This wrapper code is illustrated in **Figure 3.23**, where source code is translated to target code. The system call in the source code is converted to a jump (or procedure call) into the runtime. The runtime then executes wrapper code that copies and/or converts arguments from the guest to the host and then makes the appropriate system call on the host platform. An alternative, faster implementation would inline the wrapper code and guest system call.

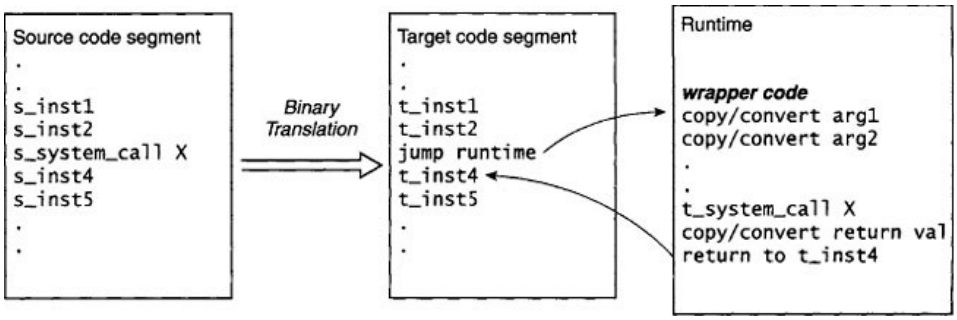


Figure 3.23 System Call Conversion When Guest and Host Operating Systems Are the Same.

Runtime-Implemented Operating System Functions

Not all guest OS operations need to be translated and passed on to the host OS. Depending on the runtime implementation and emulation method, some calls may be handled directly by the runtime. One such case is a guest OS call to establish a signal on behalf of the emulated guest application. Recall that the runtime, at the beginning of program emulation, establishes signal handlers for all signals supported by the host OS. Any exception conditions get reported to the runtime first so that it may provide the correct state to the guest process and to ensure that the runtime always retains control of the emulation process. If the application tries to establish a signal handler via a system call, the runtime handles this call di-

signal handler via a system call, the runtime handles this call directly by recording the application's signal in a side table and then returning to the guest process. Subsequently, if the guest application should trigger the signal, it first passes through the runtime before it is actually delivered to the application.

Another important area where the runtime handles guest OS calls directly is memory management. Because it essentially controls the process VM on which a guest application is running, it is the runtime that is responsible for the overall memory management. So, for example, if the guest application asks for more memory via the Linux brk() system call, then, depending on the runtime's memory management implementation, the call may be handled directly by the runtime and does not have to be passed to the host OS. The HP Dynamo technical report (**Bala, Duesterwald, and Banerjia 1999**) and Bruening's doctoral dissertation (**Bruening 2004**) contain fairly thorough discussions of the memory management functions performed by the runtime. As another example, if runtime emulation routines check memory protection via a software table (see **Section 3.4.1**) and the guest application makes a system call to change memory access privileges, then the privilege change should be emulated at the level of the runtime by modifying the software mapping table.

Nasty Realities

The discussion of OS emulation just given leaves the impression that it is a fairly straightforward process. It is not. One thing that simplifies our discussion is that we tend to focus on the Linux operating system for examples (although other versions of the UNIX OS are similar). Linux OS communication is primarily through OS calls contained in the ABI and through the signal mechanism. Even so, the foregoing discussion has been simplified.

The Windows OS is much more complex, both in the number of ways a user application and the OS communicate with each other and in the communication mechanisms themselves. Probably the

and in the communication mechanisms themselves. Probably the most thorough discussion of the problems and their solutions are contained in Bruening's doctoral dissertation (**Bruening 2004**) and in a related DynamoRIO paper (**Bruening, Duesterwald, and Amarasinghe 2001**). The major issues are briefly summarized next. A further complication when dealing with Windows is that documentation is at the API level (i.e., at the level of the user libraries that call the OS) rather than the ABI level, where the calls actually occur.

Three important "abnormal" user/kernel communication mechanisms in Windows are callbacks, asynchronous procedure calls, and exceptions. As described by Bruening, all three can be emulated in similar ways, so we focus on callbacks (**Figure 3.24**). Windows is essentially an event-driven OS; for example, a mouse click is an event that may cause input to be delivered to a user program. Events are passed to a user process (or a user thread, to be more precise) through message queues; at certain points in its execution, the OS checks a thread's message queues. If it finds an event, it calls a user routine (a callback handler) that has been registered for handling the event. This involves saving the context of the user thread and initiating a new context to run the callback handler. After setting up the callback handler, the Windows OS returns to user mode through a special dispatch routine. Then when the callback handler is finished, it normally reenters the OS kernel, where a check for further pending messages is made before returning to the original user program.

A challenge in implementing callbacks in a process VM is to maintain the runtime's overall control of the VM and to manage the execution of the callback routine. First, the runtime has to gain control before the callback handler is entered. This is done by modifying the dispatch routine by placing a jump to the runtime at the beginning of the routine. At that point, the runtime can save any of its state that is not saved by the OS during the switch to the handler (as is the

case in the DynamoRIO system). Then to detect a return from the callback, the runtime looks for a particular OS call (or a call to an API routine, which returns control to the OS); at that point, it can restore any saved runtime state prior to the switch back to the original user context.

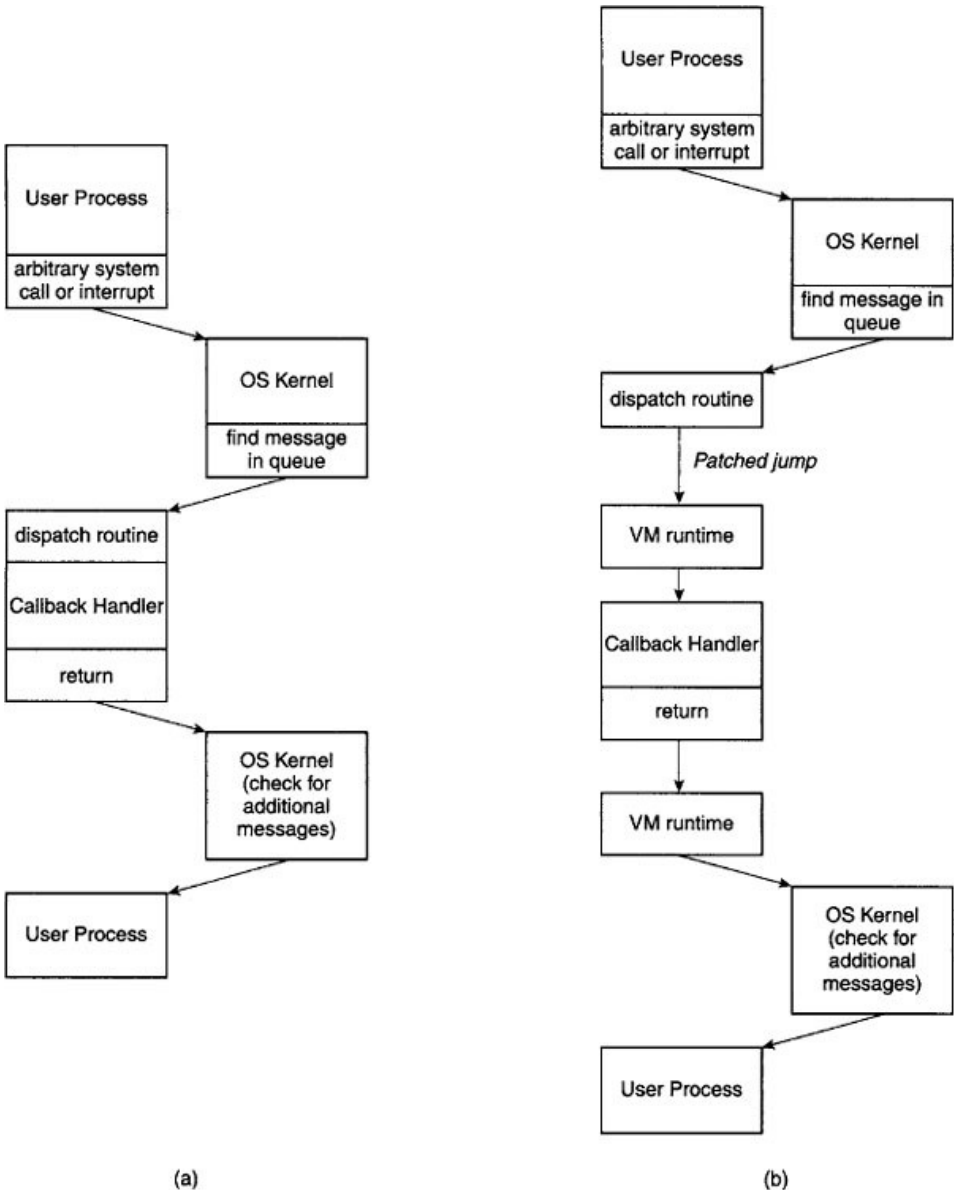


Figure 3.24 Method for Implementing Windows Callbacks. *(a) Typical Windows control flow when a message queue contains a message that invokes a callback handler, (b) To emulate and maintain control during callbacks, the runtime intercepts the transfer both to and from the callback handler.*

3.7.2 Different-Operating-System Emulation

Although it may bear some superficial similarity to instruction set emulation, emulating an OS interface is fundamentally different than emulating instruction execution. An ISA reads data from memory, executes instructions that transform the data, and writes data values back to memory. Instruction sets are *logically complete,* in the sense that any function can be performed on input operands to produce a result, given enough time. Furthermore, most ISAs perform the same basic functions: memory load/stores, arithmetic/logical operations, and branches/jumps. Hence, with ISA emulation it is not a matter of *whether* emulation can be done, just how *efficiently* it can be done and how long it will take.

In contrast, the OS deals with the outside world and with real I/O devices. Furthermore, an operating system implements and manipulates fairly complex entities (e.g., processes, threads, files, message buffers) with a well-defined set of operations. In this environment it is quite possible that the host OS simply cannot provide a function required by a guest OS, no matter how much time or human ingenuity is available. A simple example is a system call that returns the time of day. If the host OS does not maintain a time-of-day clock and the guest OS does, then no matter how much effort is put into it, the guest OS will not be able to coerce the host OS into providing the correct time of day. Similarly, the guest OS may have calls that support certain disk file operations that are simply not supported by the host. There are also semantic mismatch problems, where no combination of host operations can exactly emulate a guest operation; i.e., if a number of operations are combined, there may be additional side effects.

It is difficult to come up with a set of overall rules or strategies for different-OS emulation because of the wide variety of cases that must be handled. Consequently, OS emulation is very much an *ad hoc* process that must be implemented on a case-by-case basis and that requires considerable knowledge of both the guest and host op-

erating systems. The process is somewhat similar to porting code from one OS to another—but in some respects it is more difficult, because with porting, one can examine the original code as a whole and then make modifications using an overall strategy. With emulation, however, the conversion must be done dynamically, with relatively little context within which a particular system call can be analyzed.

In general, a guest OS call is supported by a runtime wrapper routine, as described earlier. A single guest call may require multiple host OS calls for emulation, or the runtime itself may perform some or all of the emulation rather than passing it to the host OS.

Example: Enforcing File Limits

To a large extent, both Linux and Win32 provide similar functionality with respect to file I/O. There are other cases, however, where they differ, for example, in setting file limits. When a Linux process is created it can be given an upper limit on the size of files that it can write, RLIMIT_FSIZE. There is no corresponding limit in Win32. Consequently, an interesting emulation problem occurs when a Linux guest process is run on a Win32 host. In this case, it is up to the runtime to maintain the Linux RLIMIT_FSIZE. It can do this by holding a table of file limits, with a table entry for each of the guest's open files. Then as part of the wrapper for disk write functions, the runtime can maintain the file limit table (and enforce the required limits). In addition, the Linux calls getrlimit() and setrlimit(), which read and write the limit, should be implemented directly by the runtime without calls to the Win32 host.

Practically speaking, if the guest and host operating systems are different, then there is relatively little likelihood that a completely compatible OS emulation can be performed; some compromises and approximations will be required. A compromise often used is to restrict the set of applications that can be emulated (and in the

process to restrict the system calls that must be emulated). An example is the Wabi system (**Hohensee, Myszewski, and Reese 1996**) developed by Sun Microsystems to emulate only certain widely popular Windows applications, such as Word and Excel, on Solaris platforms.

3.8 Code Cache Management

The code cache is a central part of a binary translation system, and managing the code cache is one of the important emulation-related functions performed by the runtime. Although it is similar to a conventional hardware cache in some respects, the code cache differs from a conventional cache memory in at least three significant ways.

1. The cached blocks do not have a fixed size. The size of a code cache block depends on the size of the translated target block, and this can vary from a few instructions to many tens of instructions.
2. The presence and locations of the blocks are dependent on one another because of chaining. If a translated block is removed from the code cache, then any link pointers directed toward the removed block must be modified.
3. There is no copy of the cache contents in a "backing store." After a block is removed from the code cache, it must be regenerated from the source binary image before it can be placed back into the cache.

All three of the foregoing have a significant effect on the code cache management algorithms that are used.

3.8.1 Code Cache Implementations

The code cache itself contains blocks of target code, formed by translating source code. In addition to the actual code cache, supporting side tables must be provided. Thus far in our discussion, we have identified two key operations involving the code cache that must be performed during the overall emulation process.

- Given a source PC at the beginning of a block, find the corresponding target PC

- Given a source PC at the beginning of a block, find the corresponding target PC in the code cache, if the translation exists. This is done whenever emulation control is to be transferred to the code cache via a source PC, e.g., when passing from interpretation to translated code.
- Given a target PC in the code cache, find the corresponding source PC. This is done to find the precise source PC when an exception occurs.

These two operations are performed by the map table (**Figure 2.28**) and the reverse translation side table (**Figure 3.22**), respectively.

3.8.2 Replacement Algorithms

In a practical implementation the size of the code cache is limited, and, depending on the working set size of the emulated application, it may eventually fill up with translated blocks. When this happens, some code cache management is required to make room for new translations by removing one or more of the translated blocks from the cache. Just as with conventional caches, there are a number of possible replacement algorithms. However, the differences between code caches and conventional caches cited earlier have a significant impact on the choice of replacement algorithms. We discuss a number of possible replacement strategies in the following subsections.

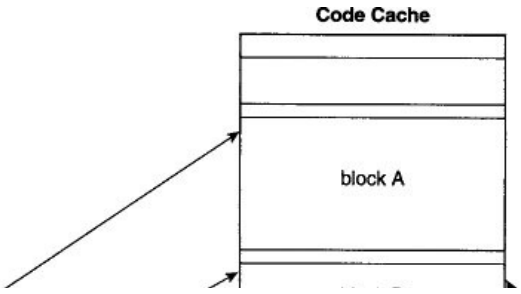
Least Recently Used

A least recently used (LRU) algorithm replaces a cache block that has gone unused for the longest period of time. Because of temporal locality, this is often a good strategy for conventional caches. Unfortunately, because of the specific properties of the code cache, an LRU cache replacement algorithm is relatively difficult to implement. First of all, there is an overhead in keeping track of the least recently used translation blocks; this would probably require a couple of extra memory accesses per block to update LRU information each time a block is entered. Second, when an arbitrary block is removed, any blocks that are linked to it must have their link pointers updated (to direct them back to the VM rather than to the removed block).

To implement delinking of blocks, *backpointers* can be added to the overall code cache structure. These backpointers can be maintained as part of the PC map table (or in a separate side table). In **Figure 3.25** an implementation is given where they are part of the map table. For each table entry (i.e., for each translation block) the backpointers, structured as a linked list, point to translation blocks that chain to the given block. When a block is replaced, the list can be followed to find and modify all the chain pointers pointing to the block.

A third problem with LRU replacement is that it may select a translated block from an arbitrary location in the middle of the translation cache, and the new block replacing it may not be exactly the same size as the block (or blocks) that it replaces. Clearly this will lead to cache fragmentation and will require mechanisms both to keep track of the sizes and locations of unused space and to perform occasional compaction of the cache (which will bring along additional complications, such as adjusting links connecting code blocks).

Finally, a somewhat lesser problem is that maintenance of the reverse-translation side table (shown in **Figure 3.22**) becomes more complicated. If this table is arranged in order of increasing target PCs, it permits a binary search (rather than a linear search) for performing a lookup in the reverse translation table. If LRU replacement is used, however, such an order cannot be guaranteed for the target PCs.



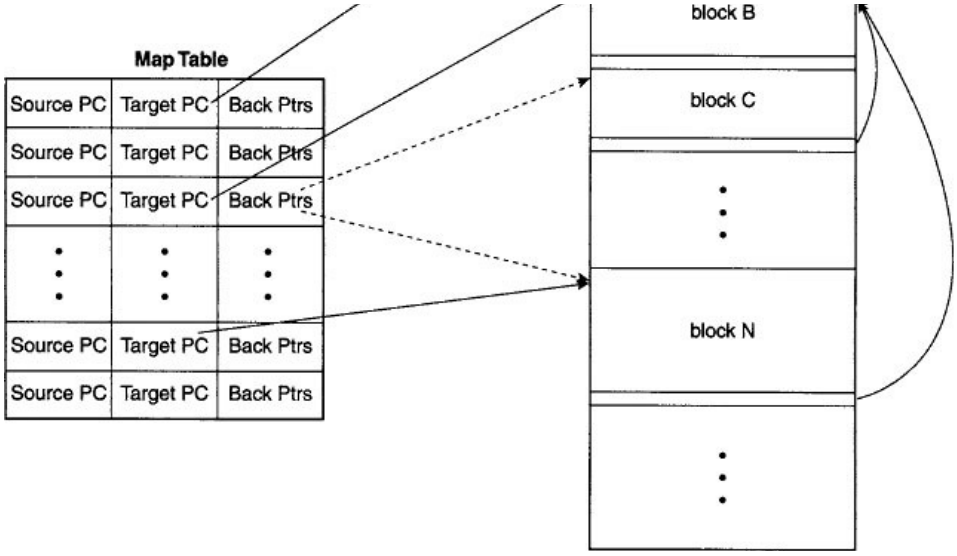


Figure 3.25 Code Cache and PC Map Table. *In this example, blocks C and N both link to block B; consequently, the map table entry for block B has backpointers to both blocks C and N.*

Because all these difficulties arise in practice, LRU, and LRU-like algorithms are not typically used for code caches. Rather, replacement algorithms that reduce (or eliminate) backpointer and fragmentation problems are preferred. The following subsections describe a number of such code cache replacement algorithms, beginning with the simplest and going to the more complex.

Flush When Full

Perhaps the most basic algorithm is simply to let the code cache fill and then to flush it completely and start over with an empty cache (Cmelik and Keppel 1994). Although this is a brute-force approach, it has some advantages. First, larger translation blocks, such as superblocks, are based on frequently followed control flow paths, i.e., the directions that conditional branches commonly take. Over time, however, the frequently followed paths may change. Flushing provides an opportunity to eliminate control paths that have become stale and no longer reflect the common paths through the code. The new translation blocks formed after flushing are more likely to be representative of the paths currently followed by the code. Second, in systems like Shade, where the translation map table is not

chained but has a fixed number of entries in each row for colliding source PCs (**Figure 2.37**), a translated block will occasionally be "orphaned" when its map table pointer is removed to make room for another. Flushing removes these orphans and reclaims their code cache space.

On the other hand, the big disadvantage of the flush-when-full approach is that all the blocks being actively used (i.e., the members of the current working set) have to be retranslated from scratch, leading to a high-performance overhead immediately after the flush.

Preemptive Flush

A more sophisticated scheme is based on the observation that many programs operate in phases. A phase change is usually associated with an instruction working set change. Consequently when there is a program phase change, a new region of source code is being entered and a larger percentage of the time is spent in block translations, as illustrated in **Figure 3.26**. By monitoring the rate at which new translations are being performed, the code cache can be preemptively flushed to make room for the translations of new working set members (**Bala, Duesterwald, and Banerjia 2000**). The number of translations after the flush will be lower than what would be needed if the cache becomes full in the middle of a phase, because retranslations of the working set are avoided.

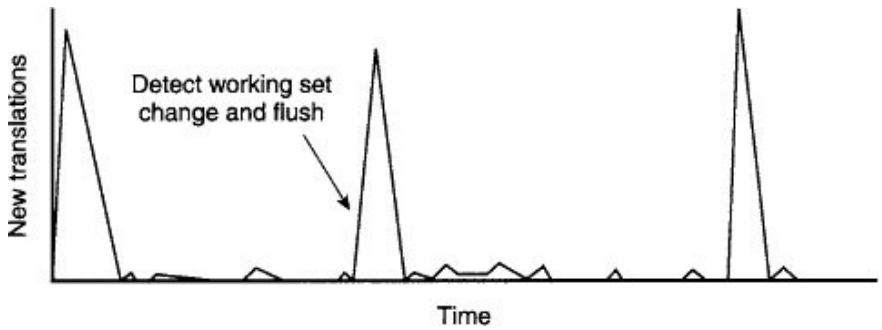


Figure 3.26 Code Cache Flushing When the Instruction Working Set Changes. *When an increase in new translations is detected, the entire code cache is flushed to make room for a new working set.*

Fine-Grained FIFO

FIFO replacement is a nonfragmenting algorithm that exploits temporal locality to some extent and is not a brute-force approach, as is complete flushing. With FIFO replacement, the code cache can be managed as a circular buffer, with the oldest block(s) being removed to make room for the newest. If the new block being inserted into the cache is of size *n*, then the set of oldest blocks whose aggregate size is at least *n* is replaced. The reverse-translation side table can be managed in a corresponding FIFO manner. A "head" pointer keeps track of the oldest translation block in the cache, and this is the point where replacements will begin.

This scheme overcomes a number of the disadvantages of LRU (albeit at a slightly reduced hit rate). However, it still needs to keep track of chaining via backpointers because individual translation blocks are replaced.

Coarse-Grained FIFO

This scheme partitions the code cache into very large blocks; for example, the blocks may be a fourth or an eighth of the entire cache. These large blocks are managed on a FIFO basis. With this scheme, the backpointer problem can be simplified or eliminated.

The backpointer problem can be simplified by maintaining the pointers only on a FIFO block basis. For example, if a translation block contained in FIFO block A links to a translation block contained in FIFO block B, then the backpointer list for FIFO block B is updated with a pointer to the translation block in A. However, if a translation block links to another block contained in the same FIFO block, then no backpointer is needed. See **Figure 3.27**. When a FIFO block is replaced, all the backpointers associated with it are followed to remove links from translation blocks coming from other FIFO blocks. Due to temporal locality in programs, the number of

FIFO blocks. Due to temporal locality in programs, the number of intra-FIFO block links, which do not have backpointers, is likely to be considerably larger than the number of inter-FIFO block back-pointers, which need to be dealt with when a FIFO block is replaced.

A coarse-grained FIFO approach was proposed in Mojo (W.-**K. Chen et al. 2000**), where only two major blocks were used, with no back-pointers between blocks. Also, note that, in a sense, complete flushing is a degenerate special case of coarse-grained FIFO (i.e., it uses only one block).

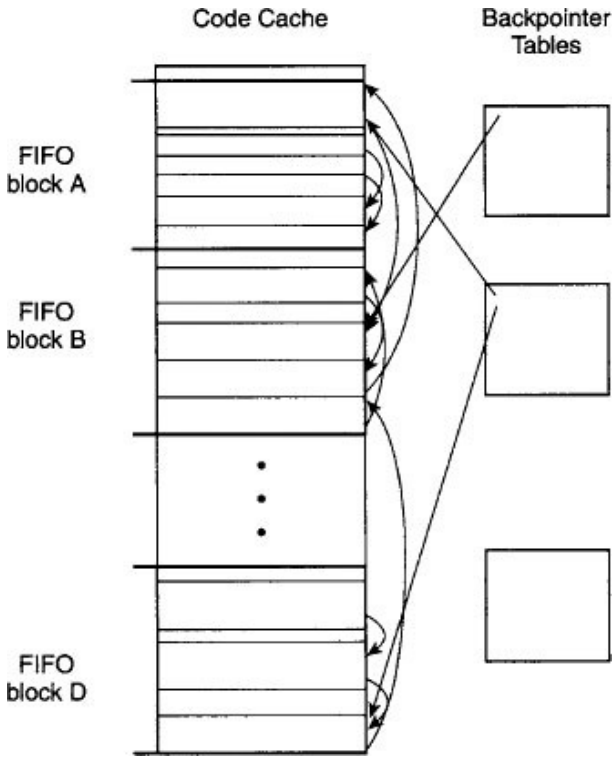


Figure 3.27 Code Cache Managed with a Coarse-Grained FIFO. *Backpointers are needed only for chain pointers that cross FIFO replacement block boundaries.*

Performance

A study of code cache replacement algorithms (**Hazelwood and Smith 2004**), based on the Dynamo/RIO dynamic optimization system (**Bruening, Garnett, and Amarasinghe 2003**), determined that a coarse-granularity FIFO scheme, e.g., dividing the cache into eight large blocks, tends to have lower overhead than either fine-grained

FIFO or flush when full. The graph in **Figure 3.28** shows overhead (relative to flush when full) averaged over a number of benchmark programs, including SPEC benchmarks and some Windows programs. The code cache size was artificially restricted for each benchmark to determine performance effects when the code cache is stressed (when it is not stressed, then the replacement algorithm makes little difference).

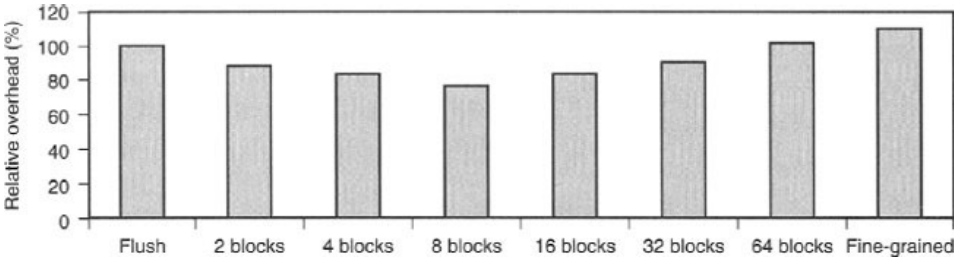


Figure 3.28 Relative Overhead of FIFO Code Cache Management Schemes. *Overheads are relative to a flush-when-full method (Flush).*

3.9 System Environment

As the last step in our development of process VMs, we consider the integration of guest processes into the host system environment. In a system that supports process VMs, it is desirable to make the installation and execution of guest applications as transparent as possible, i.e., without the users' having to take any extraordinary actions. The goal is to give the user seamless, transparent access to both host and guest applications.

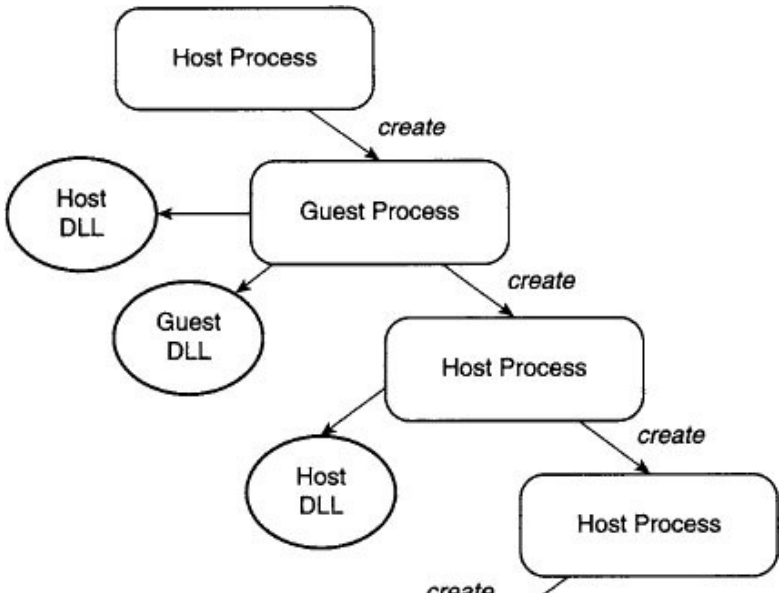
The major issue in supporting an integrated environment is the encapsulation of all guest code in an emulation environment at the time it is loaded. In general, guest code may be loaded at the time of process creation, or it may be dynamically linked later. The type of environment that is to be supported is illustrated in **Figure 3.29**. Here we see that a guest process may create either another guest process or a host process; similarly, a host process may create either a host process or a guest process. And as a guest process executes, it

can call a dynamically linked library (DLL) routine that may be a guest routine (and hence emulated), or it may call a natively coded host routine. In general, a host process will not call a guest DLL, so we do not consider this case.

The environment just described achieves a fairly high level of interoperability between guest and host processes. In some cases, especially if the guest and host operating systems are different, the level of interoperability may be less. For example, the ability for a guest process to invoke a host DLL may be impaired.

In order to implement the system illustrated in **Figure 3.29**, there must be two different loaders: one for host binaries and the other for guest binaries. There are several approaches for invoking the correct loader when a process is created. There are three possibilities (**Hookway and Herdeg 1997**) that are summarized here.

1. Modify the host's kernel loader routine (and any user space loaders that may be used on the host system), first to identify the type of binary being loaded (host or guest) and then to call the proper loader routine. This approach is conceptually straightforward but requires modification of kernel loader code. If the process VM system is to be implemented without kernel modifications, then this approach cannot be used.



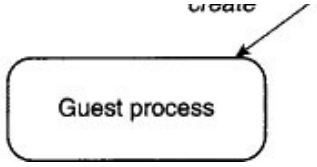


Figure 3.29 The Integration of Host and Guest Processes in a Process VM. *In a process virtual machine, host and guest processes can be created in arbitrary sequences. A host process will call only host DLLs, while a guest may call either a host DLL or a guest DLL.*

2. When a guest program binary is installed, it is converted into a host-executable file by appending some host-executable code and including the guest binary image as a large data structure. The host-executable portion of the program invokes a guest loader that loads the guest binary data structure and begins emulation. This method requires the user to identify guest binaries when they are installed so that the installer can attach the host-executable code that calls the loader frontend. In a sense, this method encapsulates a guest process at the time it is installed. This could lead to problems if guest executables are loaded from remote systems, however. That is, they must be installed within the "domain" of the process VM system.
3. Host processes are modified so that special loader code can be "hooked" on to system calls that load user processes, e.g., the CreateProcess call in Win32 or the exec() call in Linux. This method is used in FX!32 (described in more detail later), where it is called *enabling.* After a process has been enabled and it wishes to call CreateProcess, a library call is first made to a special preloader *(transparency agent* in FX!32 terminology). The preloader checks to see if a guest process is being created. If so, it calls the guest process loader; if not, it creates the host process, as requested, and then enables it, before allowing it to start running. Consequently, any later processes that the newly enabled host process attempts to create will also perform the guest/host preloader check. To get this bootstrap method started in the first place, any "root" user processes, for example, the log-in shell, must be explicitly enabled.

As noted, the first method requires kernel modification, the second requires executable modification at install time, and the third modifies processes dynamically as they run. For the third method to work, it must be possible to enable an arbitrary host process. For example, in FX!32, all the calls to CreateProcess go through a single API routine that the preloader (transparency agent) is able to locate. However, if a call to create a process can occur at an arbitrary point in a host process, then enabling it would be much more difficult. One possibility is to perform the equivalent of emulation on all host processes as well as user processes. This would allow the runtime to

processes as well as user processes. This would allow the runtime to locate all system calls that are made. Slowdown could be kept very low because "binary translation" would amount to little more than identifying and caching superblocks, with no actual translation taking place. This technique will be explained in more detail in **Chapter 8**, in the context of system virtual machines.

With respect to DLLs, we assume that host DLLs can be invoked only by host processes. Consequently it is not necessary to locate those points in a host process where a DLL may be invoked. Because guest processes are always emulated, calls to the dynamic linker can be intercepted by the runtime. The runtime can then locate the DLL being called. This can either be from a library of guest code, in which case the DLL also comes under runtime control for emulation, or it may be a DLL written in the host ISA (for faster guest process execution).

3.10 Case Study: FX!32

FX!32 was developed by Digital Equipment Corporation to enable the transparent execution of IA-32 applications on Alpha platforms running the Windows operating system (**Hookway and Herdeg 1997**; **Chernoff et al. 1998**). FX!32 provided a process virtual machine model, as described in this chapter (although it is not called a *virtual machine* in FX!32 documentation and articles). FX!32 performs staged emulation, using both interpretation and translation. But unlike other systems, it does not dynamically translate code. Rather, the developers of FX!32 decided to perform translation and optimization *between* program runs, not *during* a program run. This approach is based on the performance tradeoff between run time and translation/optimization time. By not translating and optimizing dynamically, virtually all of a program's run time can be devoted to emulation. Furthermore, because translation and optimization are done between program runs, much more time can be dedicated to

these tasks, and a more highly optimized binary can be produced.

During the initial run of a program in the FX!32 system, it is interpreted only. During interpretation, execution profile information is collected in a hash table. The profile information contains, among other things, the targets of indirect jump instructions. Then between program runs, the translator can use the profile information to discover code regions starting with jump targets and then translate and optimize the resulting code. This optimized code is stored in a database on the disk.

During subsequent runs of the program, the translated and optimized code is loaded from disk and executed. If the translated code attempts to jump to a region of code that has not been translated, then the interpreter takes over with the original version of guest code. Additional execution profile information is collected, as before, so that after the program finishes, more code can be translated and optimized. Eventually, all parts of the guest code that are executed will be translated, optimized, and stored on disk.

Conceptually, this method is similar to staged emulation with interpretation and dynamic translation. In particular, an interpreter and translator are alternately invoked, with the code being translated in an incremental fashion as it is discovered. The difference is that translated code is cached on the disk between runs, rather than being translated and held in a code cache every time it is executed. Furthermore, all discovered code is translated; it does not depend on how often the code is emulated, as in a conventional staged emulation strategy.

Using this approach increases the importance of fast interpretation, because the interpreter is used for the entire first program run and for newly discovered sections of code during successive runs. The fast interpreter is similar in structure to the one described in **Section 2.4.3**. The binary translation process is similar to that de-

scribed in **Chapter 2**, so we will not discuss it further here. Rather, we will focus on the overall FX!32 system environment.

The main FX!32 components are illustrated in **Figure 3.30**. The interpreter (called an *emulator* in FX!32) and the translator are the main parts of the emulation engine, and these have already been discussed. The other five components are the transparency agent, the runtime, the database, the server, and the manager.

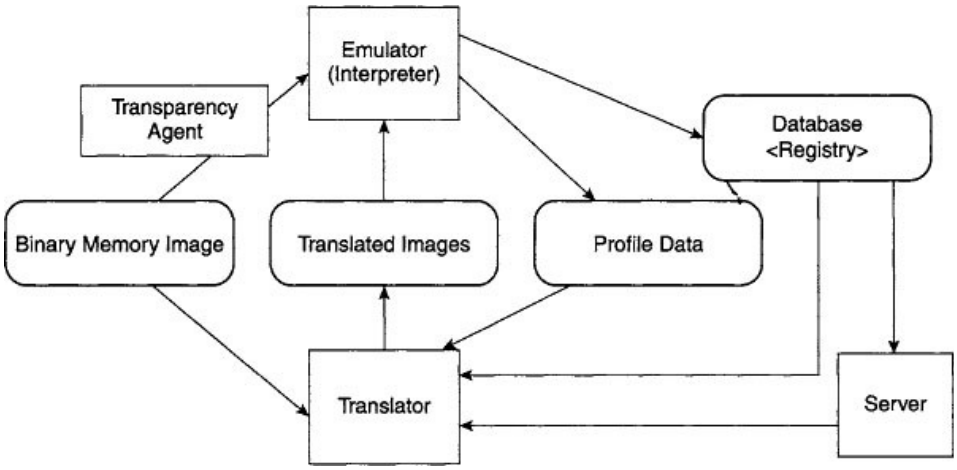


Figure 3.30 Major Blocks in the FX!32 System.

The transparency agent is responsible for providing seamless (transparent) integration of host and user processes by performing the preloading function described earlier, in **Section 3.9**. The transparency agent is a DLL that is inserted into the address space of a host user process and that hooks calls on the Win32 CreateProcess system call as well as other calls related to process creation and management. This enabling process consists of a series of Win32 system calls that first allocate a small region of memory in the host process being enabled and then copy in the agent DLL code. A small number of root processes must be explicitly enabled as part of the FX!32 installation process. These are the login shell (explorer. exe), the service control manager, and the remote procedure call server. After these are enabled, all subsequent host processes created are also enabled by the transparency agent, in a recursive manner.

The runtime, as indicated in the previous paragraph, acts as a loader for guest IA-32 processes, initializes the guest environment, and manages the overall emulation process as a guest program runs. It invokes translated code, finds the source PC values for untranslated code, and invokes the interpreter when needed. The runtime also locates all the Win32 API calls and provides jacket routines that copy arguments from the IA-32 stack into Alpha registers (and performs any data conversion that may be required). Consequently, many of the API calls are performed with the host's optimized Alpha code.

The FX!32 database contains information regarding the installed IA-32 applications. This information includes, among other things, profile data for the application as well as any translated/optimized code. When the runtime first encounters an IA-32 program, it registers the image in the database and generates a unique image identifier by hashing the image's header. Then when the runtime loads an IA-32 application, it searches the database using the image identifier. If the runtime finds a translated image, it loads the translated code along with the IA-32 image.

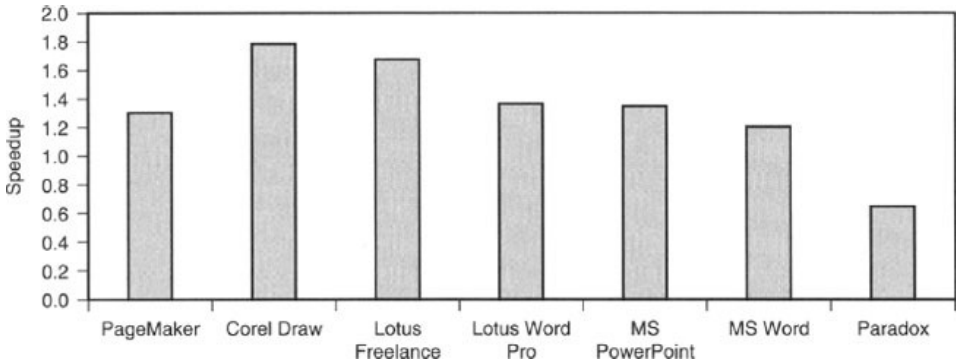


Figure 3.31 FX!32 Performance on Windows Benchmark Programs. *Performance on an Alpha 21164A is compared with a contemporary Pentium processor (Chernoff et al. 1998).*

The server is started whenever the system is booted. Its main responsibility is to automatically run the translator and optimizer between program runs. Finally, the manager provides the user some control over FX!32 resource usage. Through the manager, the user

control over FX!32 resource usage. Through the manager, the user can control the amount of disk space consumed by the database, the information that should be kept in the database, and the conditions under which the translator should be run.

Performance of the FX!32 system was quite good. **Figure 3.31** shows performance for some Windows benchmark programs (**Chernoff et al. 1998**). This data compares a 500-MHz Alpha 21164A processor with a 200-MHz Pentium Pro processor. These two processors were of roughly equivalent technology generations. Alpha performance is for binaries that have been optimized (i.e., for a second run using the same input data as the first). Performance is given as speedup with respect to the Pentium Pro, so any number larger than 1 is an improvement. The Alpha is significantly faster, with one only exception, where performance is roughly 0.65 of the Pentium Pro. The 21164A executes 4.4 Alpha instructions per IA-32 instruction, or 2.1 Alpha instructions per Pentium Pro micro-operation. The much higher 21164 clock frequency yields an overall performance improvement.

3.11 Summary

In this chapter, we considered a number of process VM implementation options. For example, instruction emulation can be done via (1) interpretation, (2) binary translation, or (3) a staged combination. Address mapping and memory protection can be done (1) via a runtime-managed table or (2) using direct mapping. The options most suitable for a given implementation depend on performance, complexity, and compatibility tradeoffs.

The most general method for implementing emulation is to use instruction interpretation with software memory mapping and protection checking. This method accommodates intrinsic compatibility, but it would likely be the slowest emulation method. On the other hand, if (1) the register state of the guest fits within the host register

hand, if (1) the register state of the guest fits within the host register file, (2) the guest memory space fits within the host space, (3) the guest page size is a multiple of the host page size, and (4) the guest privilege types are a subset of the host privilege levels, then, in general, binary translation with hardware address mapping and checking is likely to be the fastest emulation method, and it also is likely that intrinsic compatibility can be achieved.

The foregoing conditions may at first appear to be rather restrictive; however, some important practical cases do satisfy these constraints. One of the important practical cases is the virtualization of the IA-32 ABI on a RISC platform (or an Intel IPF/Itanium platform). In this case, for example, the IA-32 register file usually fits comfortably inside the host's register file, and the 32-bit IA-32 address space fits into a typical 64-bit address space.

In many other situations, process VMs can be constructed to satisfy extrinsic compatibility constraints. That is, they only provide compatibility for some, but not all, guest binaries. The most important compromise that has to be made occurs when the host and the guest support different operating systems. In this case, emulation of the guest's OS is likely to be incomplete. This would restrict the applications to those that depend on a subset of system calls or a subset of OS features. Whether compatibility is satisfied would have to be verified on a per program basis, in all likelihood.

Although the topic of this chapter was process-level virtual machines, much of the discussion was focused on emulation frameworks, including staged emulation, code caching, and memory and exception emulation. These emulation frameworks and techniques have much broader application than process VMs. What sets the process VMs apart from the others is the point at which OS calls are intercepted and emulation is performed—for a process VM this point is at the user interface to the OS. Subsequent chapters will focus on other points of emulation, specifically including emulation of

the system ISA. Furthermore, a similar emulation framework is used in same-ISA dynamic binary optimizer systems (described as part of the next chapter). And many of the same techniques are employed in high-level language virtual machines, such as the Java VM, to be discussed in **Chapters 5** and **6**.

Previous chapter
< Chapter 2: Emulation

Next chapter
Chapter 4: Dynamic Binary Optimization >

Chapter Four

Dynamic Binary Optimization

After compatibility, performance is often the most important consideration in virtual machine implementations. This chapter picks up where the previous chapter left off and focuses on improving performance of the emulation process. When compared with interpretation, merely performing simple binary translation yields a large (order of magnitude) performance gain. However, applying optimizations on translated code provides additional improvements over basic translation.

In many VMs, simple optimizations are performed just to smooth some of the "rough edges" left over from an initial binary translation. For example, a simple instruction-at-a-time translation of the first two IA-32 instructions in **Figure 2.19** (repeated here as **Figure 4.1**a) yields five PowerPC instructions. A straightforward optimization, *common subexpression elimination,* looks across the original IA-32 instruction boundaries, discovers that two different PowerPC instructions compute r4+4, so the second can be eliminated (**Figure 4.1**b).

In some VMs, more aggressive optimizations can help close the gap between a guest's emulated performance and native platform performance, although it is very difficult to close the gap completely. Nevertheless, there are a few VM applications where optimization is one of the primary reasons for building the VM in the first place. One example is codesigned VMs, to be covered in **Chapter 7**. Another example is same-ISA dynamic binary optimizers, where the goal is to improve performance on the native platform; these are discussed in **Section 4.7** of this chapter.

Optimizations include simple techniques, such as translation block chaining, covered in the previous chapter. More advanced optimizations form large translation blocks, each containing multiple basic blocks, and employ optimization techniques that span the original basic block boundaries. Some optimizations involve reordering translated instructions to improve pipeline performance. Other optimizations apply techniques from conventional compilers (as is done in **Figure 4.1**). Finally, some optimizations exploit knowledge of program-specific execution patterns, collected as part of the ongoing emulation process. These statistics regarding a program's behavior, or its *profile,* serve as a guide for making optimization decisions.

```
addl   %edx,4(%eax)
movl   4(%eax),%edx
```

(a)

```
addi   r16,r4,4      ;add 4 to %eax
lwzx   r17,r2,r16    ;load operand from memory
add    r7,r17,r7     ;perform add of %edx
addi   r16,r4,4      ;add 4 to %eax
stwx   r7,r2,r16     ;store %edx value into memory
```

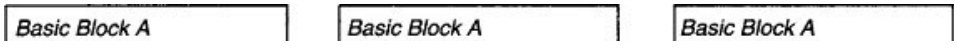
(b)

Figure 4.1 Optimization Enhances Binary Translation. *(a) A simple translation from IA-32 to PowerPC. (b) The second* addi *can be eliminated.*

Program profiling is often used to initiate shifts between emulation stages, e.g., between interpretation and binary translation, and it is also important for implementing some specific optimizations of translated code. Profile information can be collected with software by inserting instructions into an interpreter or translated code; it can also be collected by hardware or through some combination of hardware and software. Once collected, profile data is used for optimizing the performance of an emulated program. Two of the most important pieces of profiling information are (1) the instructions (or basic blocks) that are more heavily executed, i.e., where the pro-

basic blocks) that are more heavily executed, i.e., where the program spends most of its time, and (2) the sequence in which basic blocks are most commonly executed. In addition to these two principal types of profile information, specific optimizations may depend on the behavior of particular data variables or addresses, and this additional information may be collected via profiling as needed.

An advantage of run-time translation and optimization is that program profile data can provide information that may not have been available when a program was originally compiled. For example, consider the code shown in **Figure 4.2**a. Assume this represents target ISA code translated from source ISA binary code (not shown). As with other examples in this chapter, we use a register transfer notation rather than a conventional assembly language in order to make the example more easily understood. Three basic blocks (A, B, and C) have been translated; the basic block A is followed by one of the two other blocks, depending on the outcome of the conditional branch at the end of block A. Inside block A, register R1 is set to a new value that is later used in block B. If the conditional branch is decided so that block C follows the first block, then register R1 is immediately reassigned before it is used. Now let's say that profiling this piece of code indicates that the branch is predominantly decided in the direction of block C. Then the code that assigns a value to R1 in block A is useless most of the time. Consequently, an interblock optimization might remove the initial assignment to R1 from block A, as shown in **Figure 4.2**b. The resulting code will then provide improved performance because the assignment of a value to R1 is eliminated for most emulations of the original code sequence. On rare occasions, however, the branch may be decided the other way, i.e., toward block B. To deal with such an event, the optimizer must insert *compensation code* before block B is entered. In the example, the compensation code provides the proper assignment to register R1 (**Figure 4.2**c).



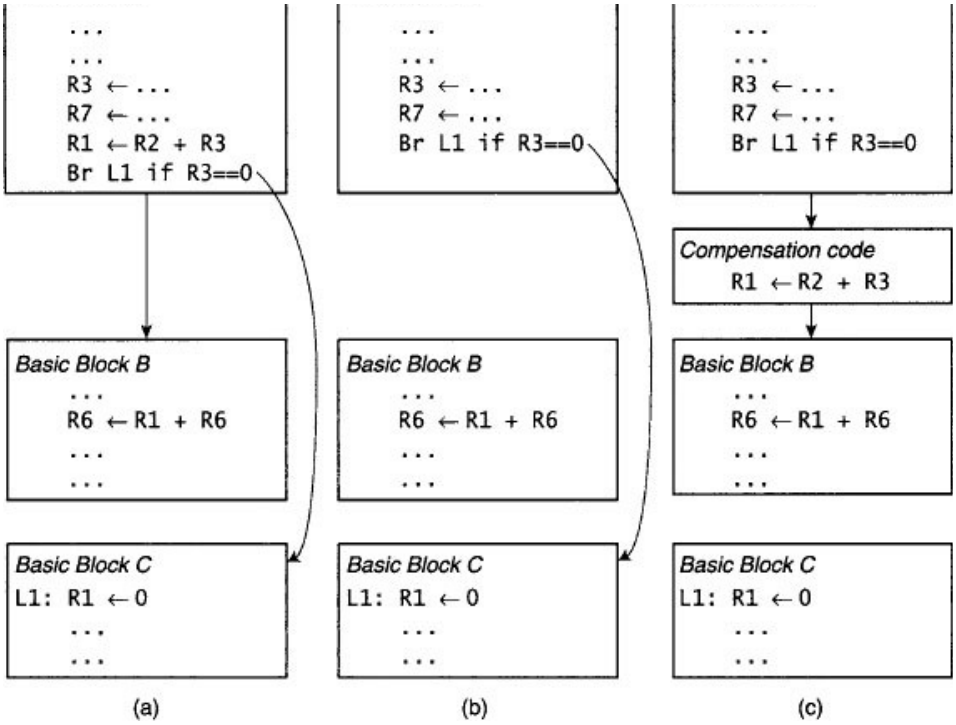


Figure 4.2 An Optimization Based on Profile Information. *(a) BEQ to basic block C is predominantly taken. (b) Optimization removes assignment of R1 from basic block A. (c) Compensation code assigning R1 is added to handle the rare branch fall-through case.*

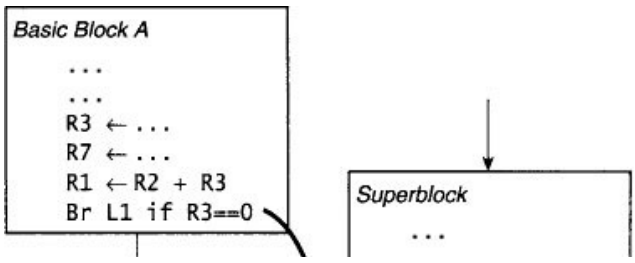
As just shown, a common optimization strategy is to use profiling to determine the paths that are predominantly followed by the program's control flow (as determined by conditional branch and jump outcomes) and then to optimize code on that basis. In some cases, these optimizations move instructions from one block to another, as was done in the example. Furthermore, some host processor implementations may rely heavily on compilers to perform code reordering in order to achieve good performance. For example, the processor may have a simple pipeline that issues instructions in order, and/or it may implement a VLIW instruction set that depends on software to combine independent instructions into a single VLIW (see Appendix Section A. 1.1). For these situations, the source program binary may not be optimally ordered for the target processor, so the optimizer performs code reordering, often with the help of profile information.

As a complement to interblock optimizations such as the one just described, the basic blocks themselves may be rearranged in memory

scribed, the basic blocks themselves may be rearranged in memory so that the most commonly followed execution path has instructions in consecutive memory locations. This is likely to improve performance by improving the efficiency of instruction fetch. If necessary, basic block rearrangement may involve the reversing of conditional branch predicates. For example, in **Figure 4.3**, the code given earlier in **Figure 4.2** is shown with the common execution path restructured as a straight-line code sequence. This straight-line sequence is an example of a *superblock*—a sequence of code with one entry point (at the top) and potentially multiple side exit points (**Hwu et al. 1993**). And, as in this example, exit points that come out of the middle of the superblock often feed into blocks of compensation code.

Most inter-basic block code optimizations do not actually require rearranging the locations of basic blocks; for example, the optimization in **Figure 4.2** does not require that a superblock be formed. However, the two types of optimizations do seem to go together naturally. The superblocks nicely localize the regions of optimization, and the localized code also provides additional performance benefits when fetching instructions, as will be discussed later in the chapter.

Optimization is tightly integrated with emulation and is usually part of an overall framework that supports staged emulation, introduced in the preceding chapter. An emulation framework contains software that coordinates multiple emulation methods and shifts among the emulation methods based on a guest program's run-time behavior.



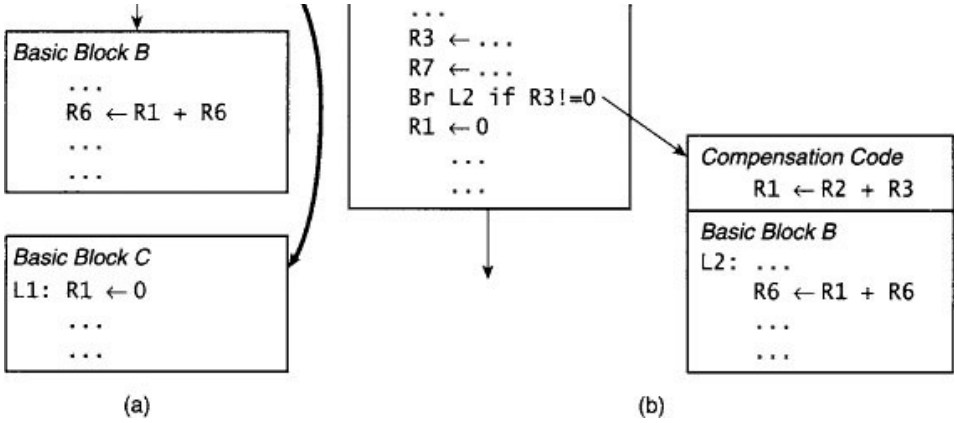


Figure 4.3 Superblock formation. *Code is rearranged so that the most frequently occurring blocks are contiguous. To accomplish this, the branch in (a) is reversed from equal to not equal. After rearrangement (b), a superblock having one entry point and two exit points is formed.*

Staged emulation, briefly described in **Section 3.5.2**, is based on the tradeoff between start-up time and steady-state performance. Some emulation methods have fast start-up performance with low steady-state performance, while other methods have slower start-up with high steady-state performance. A relatively sophisticated example of staged emulation, containing three stages, is shown in **Figure 4.4**. In this example, emulation is initially performed via interpretation (using the source ISA binary memory image). Profile information is collected concurrently with interpretation. The profile information helps identify frequently used code regions. Then these frequently executed code regions are binary translated as dynamic basic blocks and placed in a basic block cache for reuse. With the benefit of additional profile data, the frequently used basic blocks are combined into larger translation blocks, i.e., superblocks, which are then optimized and placed in the code cache. And, as emulation progresses, the optimizer may be invoked additional times to further optimize the blocks in the code cache.

To encompass the spectrum of staged emulation strategies, we can identify at least four potential emulation stages: interpretation, basic block translation (possibly with chaining), optimized translation (with larger blocks such as superblocks), and highly optimized translation, which uses profile information collected over a rela-

tively long period of time.

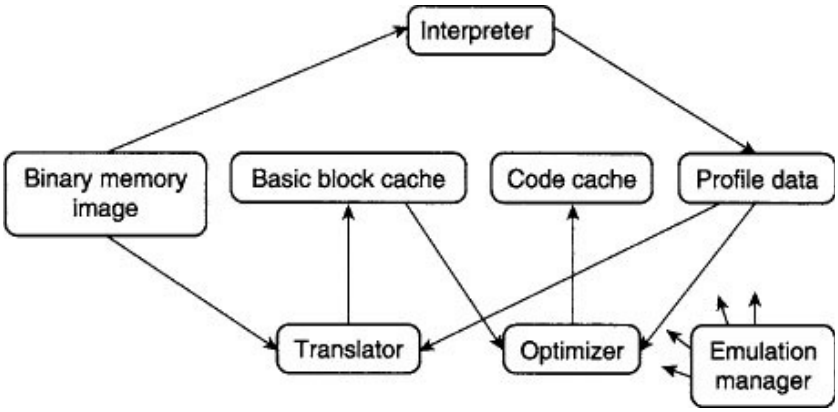


Figure 4.4 A Staged Optimization System. *Stages progress from simple interpretation to binary translation to highly optimized translation blocks.*

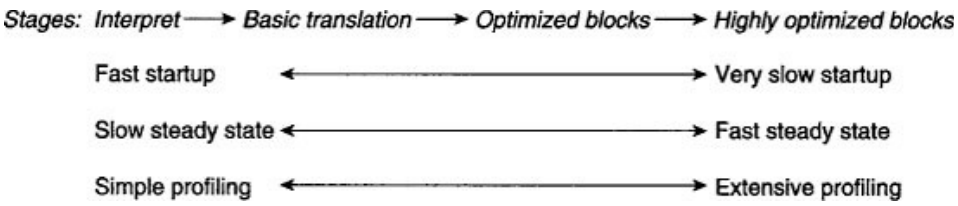


Figure 4.5 Spectrum of Emulation Techniques and Performance Tradeoffs.

Figure 4.5 shows the spectrum of emulation methods and corresponding performance characteristics. The tradeoff between start-up time and steady-state performance has already been discussed (Section 3.5.1). In addition to emulation, profiling methodologies fit across the same spectrum. For the higher levels of emulation, more extensive profiling is needed, both in types of profile data collected and the number of profile data points required. Of course, more extensive profiling may add overhead, especially if done entirely in software.

There are a number of staged emulation strategies that, depending on the source and target ISAs, the type of VM being implemented, and the design objectives. The original HP Dynamo system (**Bala, Duesterwald, and Banerjia 2000**) and the Digital FX!32 system (**Hookway and Herdeg 1997**) first interpret and then generate optimized, translated code. The DynamoRIO system (**Bruening,**

Garnett, and Amarasinghe 2003) and IA-32-EL (Baraz et al. 2003) omit the initial interpretation stage in favor of simple binary translation followed later by optimization. Other emulation frameworks omit extensive optimizations, because in some applications the overhead and/or support involved may not be justified; the optimization itself takes time, and the level of profiling required may be time consuming. The Shade simulation system (Cmelik and Keppel 1994) contains only interpretation and simple translation; in that application, optimization is not called for.

In the remainder of this chapter, the foregoing ideas are expanded upon. Profiling techniques, both software and hardware, are first described. Then we discuss optimization methods that use profile information to form large translation/optimization blocks. Next we consider optimizations that reorder the instructions within the enlarged translation blocks; for some processors, this may increase the degree of parallelism available for the underlying hardware to exploit. Then we consider other optimizations that are simple versions of classic compiler optimizations. Some of these optimizations can also take advantage of profile data collected at run time.

The optimization framework and techniques given in this chapter are targeted primarily at emulation of conventional ISAs (e.g., IA-32 and PowerPC). However, dynamic optimization is also an important part of high-performance HLL VMs (e.g., for Java), as covered in Chapter 6, and many of the techniques in this chapter can be extended to HLL VMs. The codesigned VMs covered in Chapter 7 also rely heavily on dynamic optimization. In that chapter, we focus on optimization methods that can take advantage of special hardware support provided as part of a codesign strategy.

4.1 Dynamic Program Behavior

At the time a virtual machine begins emulating a program, nothing

is known about the structure of the code being emulated, yet to improve performance the emulation software may need to apply optimizations that depend very much on both the program's structure and dynamic behavior. The optimization system can use profiling to learn about the program's structure as it is being emulated. It can then combine this profile information with typical program behavior to guide the optimization of translated code. In this section we discuss some of the important characteristics of program behavior that researchers and designers have discovered over the years. These properties underlie many of the optimization heuristics implemented in virtual machines.

One important property of programs is that dynamic control flow is highly predictable. As the program counter sequences through the program, any given conditional branch instruction is very often decided the same way (taken or not taken) a large fraction of the time. For example, consider the small piece of code in **Figure 4.6**, which searches a linear array of 100 elements for a particular value (— 1).

```
        R3 ← 100
loop:   R1 ← mem(R2)            ; load from memory
        Br found if R1 == -1    ; look for -1
        R2 ← R2 + 4
        R3 ← R3 -1
        Br loop if R3 != 0      ; loop closing branch
        .
        .
found:
```

Figure 4.6 Simple Loop That Searches for the Value—1.

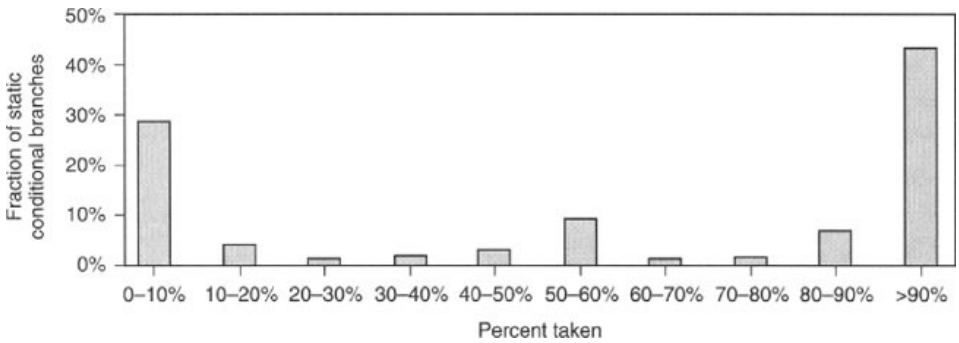


Figure 4.7 Distribution of Taken Conditional Branches. *Conditional branches are predominantly decided in one direction (taken or not taken).*

If the value of—1 appears only infrequently, then the conditional branch that closes the loop is predominantly taken. On the other hand, the branch that detects the occurrence of—1 is predominantly not taken. This bimodal branch behavior is typical in real programs. **Figure 4.7** contains average statistics for conditional branch instructions in a set of integer SPEC benchmark programs (SPEC). For these programs, 70% of the static branches are predominantly (at least 90% of the time) decided one way; 42% of the branch instructions are predominantly taken and 28% are predominantly not taken.

Furthermore, a high percentage of branches are decided the same way as on their most recent previous execution. This is illustrated in **Figure 4.8**, where 63% to 98% of branches are decided in the same direction as on their last execution, depending on the benchmark program.

Another important property of conditional branch instructions is that backward branches, i.e., branches to lower addresses, are typically taken, because they are often a part of a loop. On the other hand, forward branches are often not taken, for example, if they test for errors or other special loop exit conditions, as is the case in the example of **Figure 4.6**.

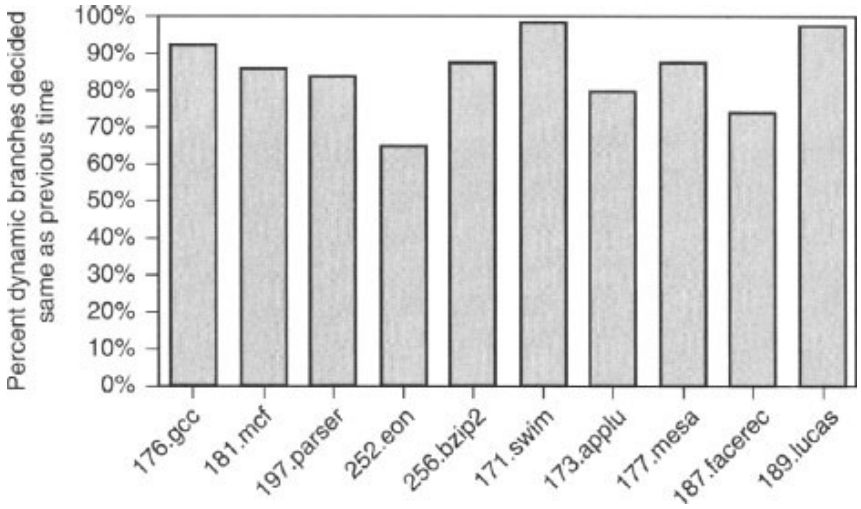
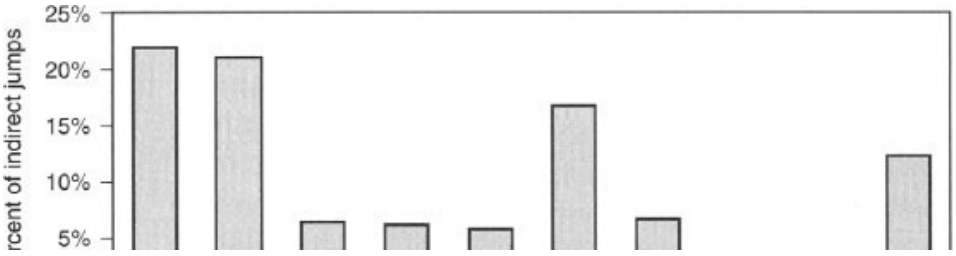


Figure 4.8 Consistency of Conditional Branches. *A high percentage of branches are de-*

cided the same way as on their immediately preceding execution.

The predictability of indirect jumps is also important for dynamic translation and optimization. With indirect jumps, the issue is determining the destination address of the jump. Because the jump destination is in a register and the register contents may be computed at run time, the jump destination may change over the course of program execution. Some jump destination addresses seldom change and are highly predictable, while others change often and are very difficult to predict. Jump instructions that implement switch statements may sometimes be difficult because the selected case may change frequently. Indirect jumps that implement the returns from procedures are sometimes difficult to predict because a procedure may be called from many different locations. **Figure 4.9** shows that for a set of SPEC benchmark programs slightly more than 20% of indirect jumps have a single destination throughout program execution, while almost 60% have three or more different destinations.

The final program characteristic in which we are interested is data value predictability. The data values used by a program are often predictable, and in many cases they change relatively little over the course of a program's execution. **Figure 4.10** shows the fractions of instructions that compute the same value every time they are executed. The leftmost bars show, for all instructions and by instruction type, the number of static instructions that always compute the same value. The rightmost bars are the fractions of dynamic instructions that execute the static instructions in the left bars. About 20% of all instructions that produce values are associated with a static instruction that always produces the same value.



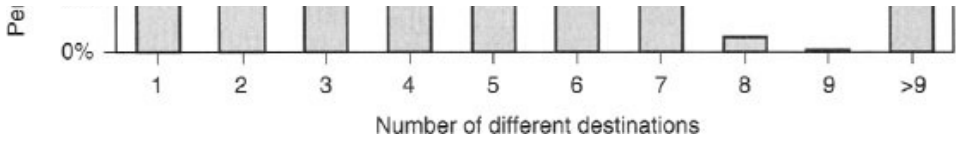


Figure 4.9 The Number of Destinations for Indirect Jumps. *Slightly more than 20% of indirect jumps have a single destination; others have a number of destinations.*

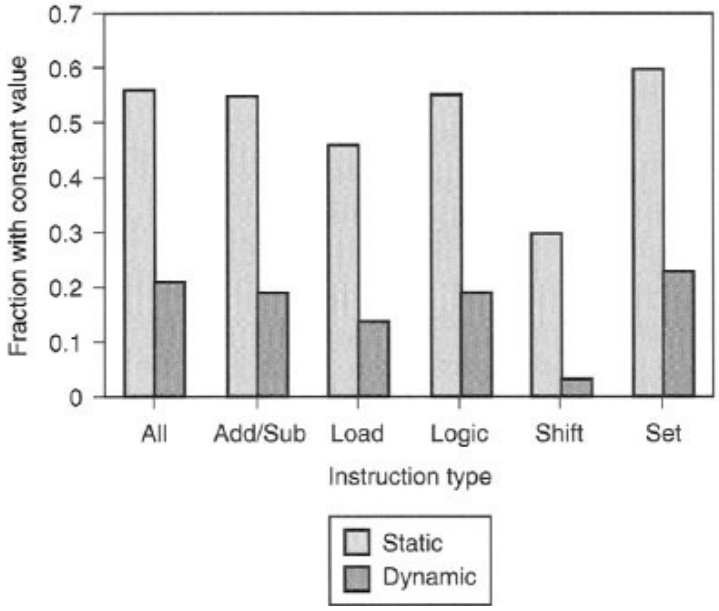


Figure 4.10 Fractions of Instructions That Always Produce the Same Result Value.

4.2 Profiling

Profiling is the process of collecting instruction and data statistics for an executing program. This statistical profile data can be used as input to the code-optimization process. In general, optimizations based on profiling work because of the predictability of programs just described. That is, program characteristics measured for past behavior will often continue to hold for future behavior and can therefore be used for guiding optimizations.

4.2.1 The Role of Profiling

Traditionally, code profiling has been used as a way of providing feedback to the compilation process under control of a software developer; refer to **Figure 4.11**a. Here, the compiler first decomposes

the source program into a *control flow graph,* then analyzes the graph as well other aspects of the program and inserts *probes* to collect profile information. A probe is a short code sequence that records execution information into a *profile log* held in memory. For example, profiling probes may be placed at the sites of branch instructions to record branch outcomes. The compiler generates code with these probes inserted. Next, the program is run with a typical data input set, and the probes generate profile data for the complete program. The profile log is then analyzed offline and results are fed back into the compiler. The compiler uses this information to generate optimized code. The optimizer may work from the original HLL program or, more likely, from an intermediate form generated by the compiler (**Chang, Mahlke, and Hwu 1991**) or the original compiled binary (**Cohn et al. 1997**). In some cases, hardware may support profile collection through either counters or timer interrupts that permit collection of statistical samples via software.

When used in this conventional setting, the program can be fully analyzed and profile probes can be optimally placed, based on the program's structure. It is not necessary to place a probe on every branch path to collect information for constructing a complete profile of branch paths. Also, a profile can be collected for an entire run of a program to gain fairly complete profile information. For improved profile coverage, multiple profiling runs with different input data sets can be used.

In contrast to the static optimization case just described, with dynamic optimization (e.g., as in the process VM shown in **Figure 4.11**b), the program structure is not known when a program begins. The runtime's emulation engine is simply given a guest binary image in memory, and both instruction blocks and overall program structure must be discovered in an incremental way, as discussed earlier in **Section 2.6**. Without an overall view of program structure, it is difficult, if not impossible, to insert profiling probes in a

manner that is globally optimal. Furthermore, profiling must be more predictive; i.e., determination of program characteristics should be made as early as possible to achieve maximum benefit. This means that optimization decisions must be made on the basis of statistics from a partial execution of the program.

There is an important performance tradeoff involving profiling overhead and the steady-state performance benefits that accrue from profiling. The profiling overhead consists of the time required for initial analysis of the program structure in order to place profiling probes, followed by the actual collection of profile data. The benefit is execution time reduction due to better optimized code. With the conventional offline profile-based optimization method, these costs are paid once, typically during later stages of program development as code is being optimized prior to release for production use. Therefore, any cost in performance overhead is experienced by the program developer in terms of longer "turnaround" time between successive profile/recompile phases. After the optimized program has been deployed to the field, however, there is no additional performance overhead, and benefits are realized every time the optimized code is run.

