result

PF: indicates parity of the least significant byte of the result.

In a straightforward emulation, every time an IA-32 add is emulated, each of the condition codes is evaluated by the emulation code. Evaluation of most of the condition codes is straightforward. For example, the sign of the result can be determined using a simple shift. In other cases, as in the evaluation of AF or PF, generating condition code values is more complex. In general, computing all the condition codes for a given source instruction takes many target instructions, often more than emulating the rest of the instruction, and it can slow emulation considerably.

It turns out, however, that although the condition code values are set frequently, they are seldom used. To make condition code emulation more efficient, a common technique is to perform *lazy evaluation,* where the operands and operation that set the condition code, rather than the condition code settings themselves, are saved (**Hohensee, Myszewski, and Reese 1996**; **Hookway and Herdeg 1997**). This allows the generation of condition codes only when needed and at the time they are needed. For example, one can maintain a lazy condition code table that has an entry for each condition code bit. The entry contains the opcode of the most recent instruction to modify the condition code bit, its operands, and its result. However, the condition code itself is not generated. Then if a later instruction, e.g., a conditional branch, needs to use one or more condition code bits, the condition code table is accessed and the needed condition code(s) are generated.

For example, the IA-32 add instruction modifies all the condition code bits. If an add operates on two registers containing the values 2 and 3, all entries in the table will contain: add: 2: 3: 5 after the instruction completes. Then if a later instruction needs to test the SF (sign), the SF entry of the table is consulted, and the result field (5) is

used to generate the sign (0). Because many instructions modify all condition code bits or because some subset of the bits are always modified together, there are various tricks that can be used to reduce the number of condition code table entries and table updates to optimize for the common case.

During binary translation, the translator can analyze sequences of instructions to determine whether implicitly set condition codes will actually be used. For example, in **Figure 2.35** there are two consecutive add instructions followed by a jmp. In this example, the condition codes set by the first add instruction are not used and do not have to be generated. This dataflow analysis can be performed as part of an overall optimization process (**Section 4.4**). There still may be cases where it is not known whether condition codes will be needed, and in these cases lazy evaluation can be used as described earlier. Also, to support efficient lazy evaluation, it may be advantageous to reserve target registers for holding condition code information, at least for the common situations.

Returning to the example in **Figure 2.35**, at the time the jmp is translated, it may not be known whether the condition codes set by the second add instruction will be needed. For example, the jmp instruction and its destination (at label 1) may be translated separately and are in two different translation blocks. Here, registers (r25–r27) are used for lazy condition code evaluation, and operand values and opcode information are saved in the registers. In this example, the code at the jmp destination address does, in fact, test the condition codes. Consequently, there is a branch to the condition code emulation routine, genZF, in order to set the ZF condition code flag. The genZF routine performs a multiway jump based on the opcode (add) and then evaluates the ZF condition code.

There are still problems, however, because it is possible for a trap to occur during emulation, and the precise (source) state, including all the condition codes, must be materialized at that point. In the fore-

going example, the first add instruction could cause a memory protection fault when it loads its operand from memory. If this occurs, the condition codes (or the lazy equivalent) must be available. In general, there must be some way to produce the condition codes for any potentially trapping instruction at the time a trap occurs, although in this situation performance is usually not an issue. Methods for materializing the correct condition code state when a trap occurs are discussed in **Section 4.5.2**, where the general problem of trap and interrupt emulation is described in the context of optimized code.

```
      addl   %ebx,0(%eax)
      add    %ecx,%ebx
      jmp    label1
             .
             .
label1:
      jz     target
```

(a)

```
      r4 ↔ %eax     IA-32 to
      r5 ↔ %ebx         PowerPC
      r6 ↔ %ecx             register mappings
             .
             .
      r16 ↔  scratch register used by emulation code
      r25 ↔  condition code operand 1     ;registers
      r26 ↔  condition code operand 2     ;  used for
      r27 ↔  condition code operation     ;    lazy condition code emulation
      r28 ↔  jump table base address

      lwz    r16,0(r4)           ;perform memory load for addl
      mr     r25,r16       ;save operands
      mr     r26,r5              ;    and opcode for
      li     r27,"addl"          ;        lazy condition code emulation
      add    r5,r5,r16           ;finish addl
      mr     r25,r6              ;save operands
      mr     r26,r5              ;    and opcode for
      li     r27,"add"           ;        lazy condition code emulation
      add    r6,r6,r5            ;translation of add
      b      label1
             .
             .
label1:
      bl     genZF               ;branch and link to evaluate genZF code
      beq    cr0,target          ;branch on condition flag
      .
      .
genZF
      add    r29,r28,r27         ;add "opcode" to jump table base address
      mtctr  r29                 ;copy to counter register
      bctr                       ;branch via jump table
      .
      .
"sub":
```

```
      .
      .
"add":
      add.   r24,r25,r26          ;perform PowerPC add, set cr0
      blr                         ;return
```

(b)

Figure 2.35 Lazy Evaluation of IA-32 Condition Codes with Binary Translation. *(a) Intel IA-32 code sequence; (b) PowerPC translation using lazy condition code evaluation.*

Finally, even if the target ISA uses condition codes, they may not be entirely compatible with the source ISA condition codes (**May 1987**). For example, the SPARC ISA has conditions codes N, C, Z, V, which are equivalent to the IA-32 SF, CF, ZF, OF, but it does not have codes corresponding to AF and PF. Consequently, emulation is simplified for some of the condition codes (provided the condition code register of the target can be easily read/written), but more expensive emulation is still needed for the other condition codes.

2.8.3 Data Formats and Arithmetic

Emulating most of the instructions that transform data (e.g., adds, logicals, shifts) is fairly straightforward. The problem is simplified because data formats and arithmetic have become more or less standardized over the years. For integers, the two's complement representation has found universal acceptance; for floating-point, the IEEE standard is commonly implemented.

If both the source and the target ISAs implement two's complement arithmetic, the emulation of arithmetic operations is easily done, especially when identical data widths are supported in both ISAs. Most ISAs offer a basic set of shift and logical instructions that can be used to compose different variations of shift and logical instructions in a source ISA.

Although the IEEE floating-point *format* is commonly used, there are some differences in the way floating-point *arithmetic* is performed on different implementations. For example, the IA-32 uses 80-bit intermediate results, unlike most other ISAs. This means that the pre-

cision of IA-32 intermediate results will differ from an ISA that uses 64-bit intermediate results. It is possible, but quite cumbersome, for a non-IA-32 target ISA to perform IA-32 emulation and obtain identical results. As another example of arithmetic differences, the PowerPC ISA provides combined multiply-add instructions, which cannot always be emulated exactly by using the obvious simple sequence consisting of a multiply instruction followed by an add instruction because the intermediate precision of the multiply add may be higher than that required by the IEEE standard. Once again, emulation is possible though cumbersome.

There are often cases where the source ISA requires a functional capability not available in the target ISA. For example, some instruction sets provide integer divide instructions, while others provide more primitive shift/subtract “divide step” instructions or instructions to convert and use the floating-point divider. As another example, some ISAs implement a large variety of addressing modes, while others have a smaller set of simple addressing modes. Invariably the simpler ISA has sufficient primitives to implement the more complex instructions in the other ISA. For example, an auto-incrementing load can be done with a combination of an add instruction and a load.

Finally, immediate data can cause slight emulation difficulties because different ISAs often have different immediate lengths. Mapping shorter immediate values to longer ones is obviously easier than mapping long immediate values to shorter immediate fields. However, all ISAs have some way of building full-length constants using a small number of instructions, so all immediate values can be handled, regardless of length.

2.8.4 Memory Address Resolution

In general, different ISAs can access data items of different sizes. For example, one ISA may support loading and storing of bytes, half-

examples, the ISA may support loading and storing of bytes, half-words (16 bits), and full words, while another may only support bytes and words. Multiple instructions in the less powerful ISA can invariably be used to emulate a single memory access instruction in the more powerful ISA.

Most ISAs today address memory to the granularity of individual bytes. However, if a target ISA does not, then emulating a byte-resolution ISA on a machine with word resolution requires shifting out (and saving) the low-order byte address bits when performing memory access and then using the saved byte offset bits to select the specific bytes (or halfwords) being accessed. Performing a store of a data item less than a full word in size requires first performing a word load, inserting the data item into the word (via shifts and mask operations), and then performing a word store. The opposite problem of emulating a word-addressed ISA on a byte-addressed target is very straightforward; it is a case of emulating a less powerful ISA on a more powerful one.

2.8.5 Memory Data Alignment

Some ISAs align memory data on “natural” boundaries and some do not. That is, a word access must be performed with the two low-order address bits being 00, and a halfword access must have a 0 for the lowest-order bit. If an ISA does not require addresses to be on natural addresses, then it is said to support “unaligned” data. A conservative approach is to break up word or halfword accesses into byte accesses. Usually if an ISA does not support unaligned data directly, however, it has supplementary instructions to simplify the process, and these can be used. Typically, as a safety net, the ISA also specifies a trap if an instruction does attempt an access with an unaligned address.

Run-time analysis of the target code may help in reducing code expansion by substituting word accesses for aligned cases. In some sit-

uations it may be effective to use run-time profiling information to reduce the dynamic cost of emulating such accesses. This will be discussed further in **Section 4.6.4**.

2.8.6 Byte Order

Some ISAs order bytes within a word so that the most significant byte is byte 0 (and in a 32-bit word, the least significant is byte 3). This is referred to as the *big-endian* byte order (**Cohen 1981**). Other ISAs order bytes in the *little-endian* byte order, which addresses the least significant byte as 0 and the most significant as byte 3. For example, the character string "JOHN" is stored as JOHN in a big-endian machine and as NHOJ in a little-endian machine. In either case, if the bytes are accessed in the sequence 0, 1, 2, 3, then the string is accessed in order J, O, H, N.

It is common to maintain the guest data image in the same byte order as assumed by the source ISA. Consequently, to emulate a big-endian source ISA on a little-endian target ISA (or vice versa), the emulation code can modify addresses when bytes (or halfwords) are accessed from the guest memory region. For example, consider a load byte instruction. To get the correct data, the low-order two-address bits can be complemented (i.e., to convert byte address 00 to 11, 01 to 10, etc.). As a more complicated example, if an unaligned word address is used, then the bytes can be individually loaded from addresses found by sequencing through individual byte addresses and complementing the addresses before accessing memory. That is, if the source ISA specifies address xxx01001, then the bytes to be loaded are at locations xxx01010, xxx01001, xxx01000, xxx01111. Or, alternatively, a pair of load words combined with shifts and logical instructions can be used to assemble the correct data in the register of the target. In any case, it is an awkward, time-consuming process that is hard to avoid, in general. Some ISAs support both endian orders (via a mode bit), and a target ISA with this feature would clearly simplify the emulation process.

Finally, note that byte-order issues extend into the host operating system calls. That is, guest data accessed by the host operating system also has to be converted to the proper byte order. This can be done in the “jacket” or “wrapper” code that supports OS call emulation (see **Section 3.7**).

2.8.7 Addressing Architecture

The memory addressing architecture is a very important part of an ISA, and it poses a number of difficult problems when emulating full applications. The address space sizes of the source and target ISAs may be different, and/or their page sizes may be different or their privilege levels different. These address space issues are fairly complex, and often the virtual machine architecture as a whole must be considered in order to provide a good solution; that is, it is not a matter of instruction emulation alone. Because the virtual machine as a whole provides a proper address space environment, this problem is deferred to **Chapter 3**, which discusses specific virtual machine implementations.

2.9 Case Study: Shade and the Role of Emulation During Simulation

Simulation is a key part of computer system design. Simulators are used for studying the performance of memory hierarchies and the internal behavior of superscalar microarchitectures. They are also useful for developing compiler optimizations and debugging. Simulators typically operate on benchmark programs or kernels and emulate these programs as a part of the simulation process. That is, a simulator includes emulation, but it does more. In simulation, the objective is to study the *process* of performing computation, not the computation itself; that is, one is seldom interested in

the actual output produced by a simulated program (especially when the program is being simulated for the 100th time with exactly the same input data!). In addition to emulating a program, a simulator may model the internal workings of a processor or memory system. It may directly track the operation of hardware features, or it may generate instruction or memory address traces for further evaluation by other simulation tools. Often performance is sacrificed in order to collect process-related information, such as branch misprediction accuracy or numbers of instructions issued on particular cycles.

Shade is a simulation tool developed for high-performance simulation (**Cmelik and Keppel 1994**, **1996**); consequently, it contains a sophisticated emulator that forms a basis for flexible and extensible simulation. It does this by providing a set of functions that can be linked into the emulator. Some of the functions are provided as part of the Shade toolset, or the user can write additional functions.

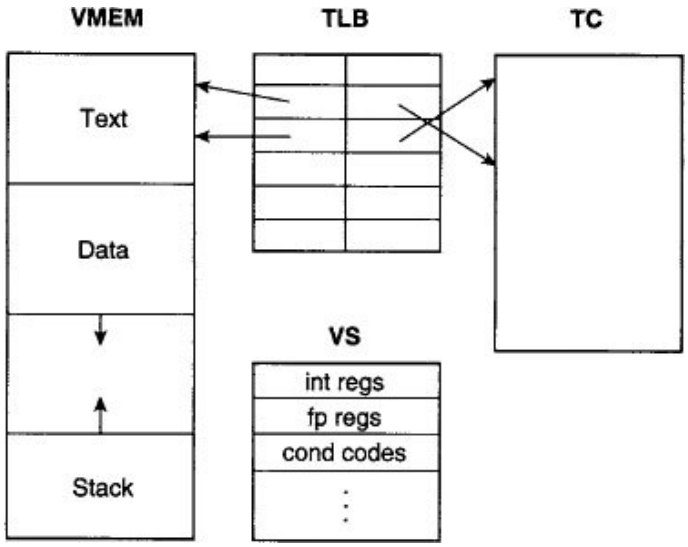


Figure 2.36 Primary Data Structures in Shade. *The source memory image (VMEM), the source register state (VS), the cache for holding translated instructions (TC), and the table mapping source to target PCs (TLB).*

Shade first translates the ISA being simulated (what we have been referring to as the source ISA[3]) into sequences of target instructions. In general, basic block-size units are translated. The translated code

not only performs emulation but also generates simulation trace data via function calls embedded in the translated code. Because we are interested primarily in emulation, we skip the details of trace generation, but the reader can refer to the excellent articles describing Shade cited earlier.

Figure 2.36 gives the important elements of Shade. The source ISA memory image is kept in a large array VMEM. The register state, including the condition codes, and control registers are held in the table VS. A trace cache (TC) is the code cache mechanism that holds translated blocks of target instructions. Finally, the mapping between source PC values and target PC values is maintained in the *translation lookaside buffer,* or TLB.[4]

To speed up emulation, the source program counter (VPC) and the base of the source memory image (VMEM) are permanently mapped to target registers. A pointer to the base of the TLB is also permanently mapped to a target register. While executing a block of translated code, source register values (held in the VS) used in the block are temporarily copied into target registers. As they are computed, result register values are copied back into the VS.

Translated blocks in the TC may be chained together to avoid TLB lookups. The translation cache is managed in a simple way: Translations fill the TC linearly as they are produced. When the TC becomes full, it is simply flushed, and translation begins filling it again. This method is clearly simple—it avoids having to "unlink" translation blocks, and the TC is generally large enough that it does not reduce performance significantly.

The structure of the Shade TLB is shown in **Figure 2.37**. It is structured as a two-dimensional array, where a source PC hashes into a row, and each row contains *n* <source, target> pairs. The lookup algorithm linearly scans the row until it has a match (and finds the target PC) or until it hits the end of the array. If it hits the end, it as-

sumes there is no translation and forms one. The inner loop of the emulation code checks the first translation in the list. If there is a hit, it proceeds; otherwise, it calls a routine that does the linear search. This search routine then places a matching entry in the first position, so the most recently used is always the first one checked. If all *n* entries are full, then a new entry pushes the rightmost entry off the end and it is lost. This means there may be some "orphan" translations in the TC, and a redundant retranslation may later be added to the TC. However, the occasional flushes of the TC clear out the orphans. Finally, a further optimization is to align each of the TLB rows to a cache line boundary so that if there is a miss during the initial lookup, the remaining entries will be brought into the cache when the initial miss is serviced.

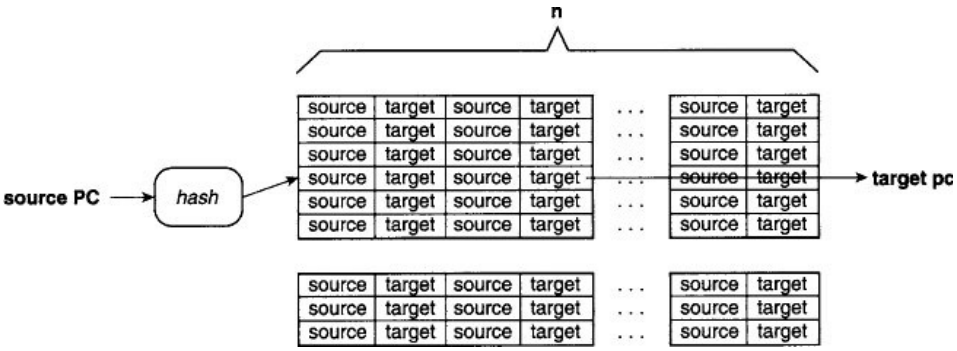


Figure 2.37 Shade TLB Structure.

2.10 Summary: Performance Tradeoffs

Figure 2.38 summarizes all the emulation methods we have discussed, and we now compare their characteristics. As criteria for comparison, we consider start-up time, memory requirements, steady-state performance, and code portability. We provide qualitative performance characteristics; a more quantitative evaluation of interpreter performance was done by **Romer et al. (1996)**. The relative importance of the criteria depends on the VM being imple-

mented. For example, memory requirements may be important in a VM in an embedded application but not in a server application. Portability may be important in HLL VMs but not in codesigned VMs.

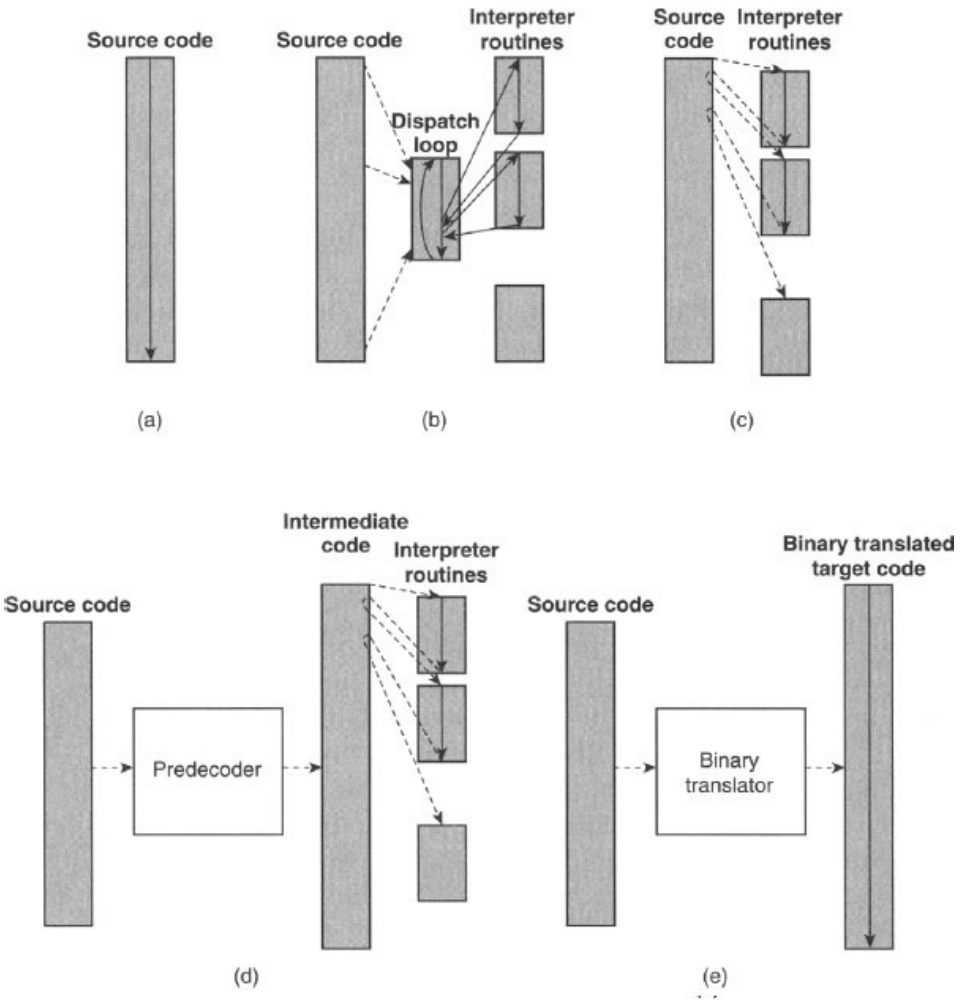


Figure 2.38 Summary of emulation Methods. *(a) No emulation; (b) decode-and-dispatch interpretation; (c) indirect threaded interpretation; (d) predecoding and direct threaded interpretation; (e) binary translation.*

Decode-and-Dispatch Interpretation

Memory requirements: low—There is one interpreter routine for each instruction *type* in the target ISA.

Start-up performance: fast—There is essentially zero start-up time because no preprocessing or translation of the source binary is re-

quired.

Steady-state performance: slow—A source instruction must be parsed each time it is emulated. Furthermore, the source code must be fetched through the data cache, which puts a lot of pressure on the cache (and leads to a potential performance loss). Finally, this method results in a high number of control transfers (branches).

Code portability: good—If the interpreter is written in an HLL, it is very portable.

Indirect Threaded Interpreter

Memory requirements: low—More memory is required than with decode-and-dispatch interpretation because the dispatch code sequence must be included in every interpreter routine. The amount of extra memory depends on the complexity of decoding; for a RISC ISA it would be relatively low, and for a CISC it could be much higher. The memory cost can be mitigated via hybrid implementations.

Start-up performance: fast—As with decode-and-dispatch interpreters, there is essentially zero start-up time; no preprocessing is required.

Steady-state performance: slow—This is slightly better than decode-and-dispatch because several branch instructions are eliminated. There is high data cache usage, as in decode-and-dispatch interpreters.

Code portability: good—The interpreter code is as portable as with decode-and-dispatch interpretation.

Direct Threaded Interpreter with Predecoding

Memory requirements: high—The size of predecoded memory im-

age is proportional to the original source memory image (and is probably larger). Memory requirements can be somewhat reduced if the intermediate form is cached and blocks of seldom-used predecoded instructions are removed from the cache.

Start-up performance: slow—The source memory image must first be interpreted in order to discover the control flow. Also, generating the decoded intermediate form can be time consuming.

Steady-state performance: medium—This is better than indirect threading because the individual instructions do not have to be parsed (and decoded) every time they are executed. If the predecoded form contains target addresses of interpretation routines, then the dispatch table lookup is eliminated. There is high data cache usage because the predecoded instructions are still treated as data by the interpreter code.

Code portability: medium—If the predecoded version contains specific locations for interpreter routines, then the interpreter becomes implementation dependent. Compilers that support finding the address of a label may mitigate this disadvantage.

Binary Translation

Memory requirements: high—The size of predecoded memory image is proportional to the original source memory image. As with predecoding, memory requirements can be reduced if blocks of translated code are cached.

Start-up performance: very slow—The source memory image must first be interpreted in order to discover the control flow; then translated binary code must be generated.

Steady-state performance: fast— The translated binaries execute directly on hardware. Performance is improved even more if trans-

lated blocks are linked directly together. Furthermore, pressure on the data cache is reduced because translated code is fetched through the instruction cache.

Code portability: poor—Code is translated for a specific target ISA. A new translator (or at least the code-generation part) must be written for each target ISA.

---

1. PowerPC actually numbers the most significant bit (msb) 0; we use the convention that the msb is 31.

2. Because the source and the target use the same ISA, we put translation in quotes; however we are using the same process as is used when true translation is done.

3. What we refer to as the *source* and the *target* are called the *target* and the *host* in Shade documentation.

4. This TLB should not be confused with the translation lookaside buffer used in many processors for caching virtual address translation information.

Previous chapter
< Chapter 1: Introduction to Virtual Machines

Next chapter
Chapter 3: Process Virtual Machines >

Chapter Three

Process Virtual Machines

A typical computer user works with a large number of programs that coexist in a system environment consisting of one or more processors, memory, a file system, and a number of peripheral devices. The user invokes and interacts with the programs by employing the interface supported by libraries and the operating system. An important restriction, however, is that the user can run only programs that have been compiled for the user's operating system and the processor's instruction set. One of the important applications of virtual machines is to circumvent this restriction and allow the user to run programs compiled for other systems. As we shall see, a number of VM architectures can provide this capability, but the simplest VM approach from the user's perspective, and the one described in this chapter, is to provide a virtual environment at the program, or *process, level.* By using a process VM, a guest program developed for a computer other than the user's host system can be installed and used in the same way as all other programs on the host system; the user as well as the other programs interact with the guest program in the same way as they interact with native host programs.

Computer programs are compiled, distributed, and stored as executable binaries that conform to a specific *application binary interface,* or ABI, which includes features of both the hardware instruction set and the operating system. For example, a very widely used ABI is the one designed to execute on a processor supporting the Intel IA-32 ISA and the Microsoft Windows operating system. A few years ago, Intel developed a new, 64-bit ISA, now called IPF, which is implemented in the Itanium family of processors. Existing IA-32 application programs will not run directly on Itanium platforms, even though the Windows operating system has been ported to Itanium

though the Windows operating system has been ported to Itanium platforms. So to allow users to run the large number of existing IA-32/Windows applications on Itanium platforms, the people at Intel developed a *virtual* IA-32/Windows environment. The resulting process VM, the IA-32 EL (execution layer), allows IA-32 programs to appear to an Itanium user exactly as they would on a native IA-32 platform (**Baraz et al. 2003**). Another version of the IA-32 EL process VM with a different OS interface supports IA-32/Linux applications.

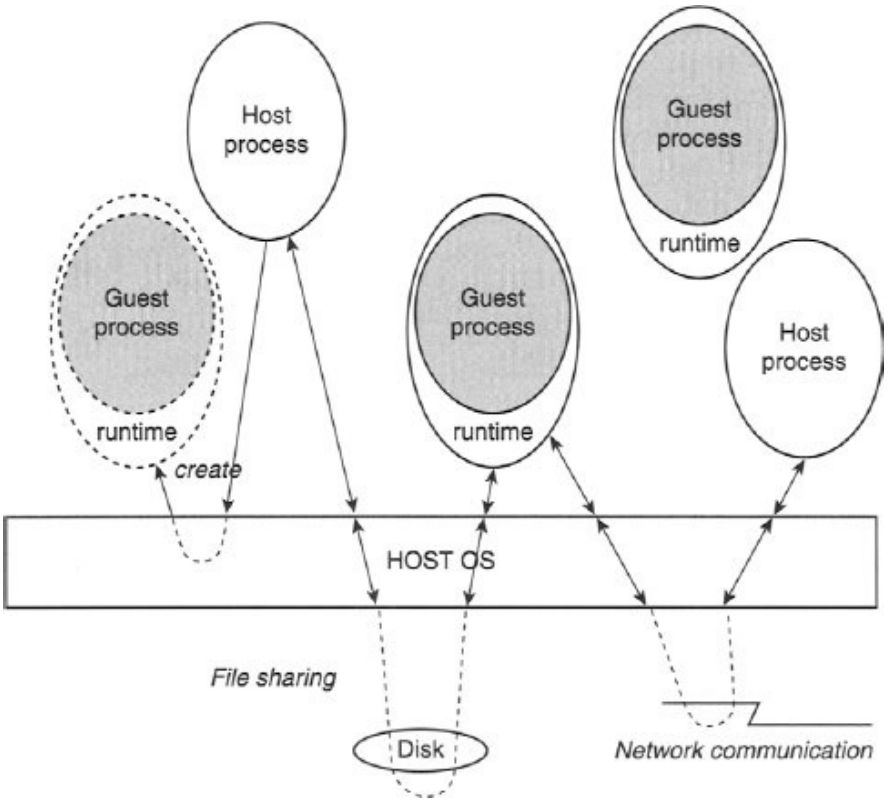


Figure 3.1 A Guest Process, with Process VM Support in the Form of Runtime Software, Interacting with Host Processes.

Figure 3.1 illustrates a typical process virtual machine environment. As shown in the figure, *runtime* software essentially encapsulates an individual guest process, giving it the same outward appearance as a native host process. From the perspective of the guest process, all the other processes appear to conform to its worldview. Consequently, the guest process can interact with the native host processes in the same way the host processes interact with each

processes in the same way the host processes interact with each other. Furthermore, the guest process can interact with other guest processes as though they were running on a real machine.

In this chapter, we discuss process VM implementations in a more or less top-down fashion. In the next section, we discuss the overall structure of a process VM. Then there is a section discussing compatibility issues, followed by sections on each of the major aspects of a process VM. These include the mapping of guest state to host state and emulation of the memory-addressing architecture, instructions, exceptions, and operating system calls. Also related to process VM implementation is a section discussing code cache management techniques. Then there is a section on the integration of a process VM into a host environment; this includes process VM loading and initialization. The chapter concludes with a case study of the Digital FX!32 system, a VM that supports the IA-32/Windows ABI.

3.1 Virtual Machine Implementation

The major computation blocks and data structures of a process VM are shown in **Figure 3.2**. The major blocks perform the following functions.

- The *loader* writes the guest code and data into a region of memory holding the *guest's memory image* and loads the runtime code. Although the memory image contains the guest's application code as well as data, it is all data as far as the runtime is concerned because source code is not directly executed. Rather, the source code is used as input "data" to interpretation and/or binary translation routines.
- The loader then turns control over to the *initialization* block, which allocates memory space for the code cache and other tables used during the emulation process. The initialization process also invokes the host OS to establish signal handlers for all the trap conditions that can occur (at least those for which signals are supported by the host OS). After initialization, the emulation process can begin, typically using a range of emulation techniques in a staged manner.
- The *emulation engine* uses interpretation and/or binary translation to emulate the guest instructions using the basic methods in **Chapter 2**. If binary translation is

guest instructions using the basic methods in Chapter 2. If binary translation is used as an emulation method, translated target code is held in a *code cache.* The code cache, which is written by the translator, is an executable code area while emulation is in progress. If emulation is performed via interpretation with a predecoded intermediate form, the predecoded instructions are stored in a similar cache structure.

- Because of the limited size of the code cache, the *code cache manager* is responsible for deciding which translations should be flushed out of the cache to make room for new translations as they are generated.
- A *profile database* contains dynamically collected program information that is used to guide optimization during the translation process. The uses of profile data, as well as optimization methods, are described in more detail in **Chapter 4**.

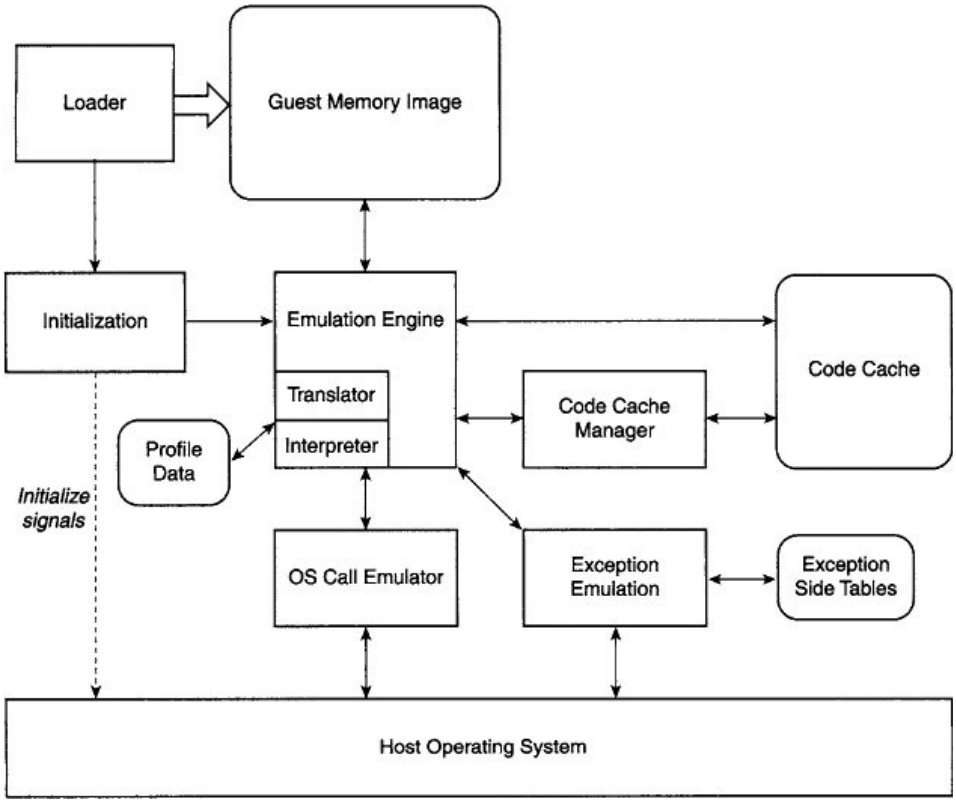


Figure 3.2 Implementation of a Process Virtual Machine.

- As emulation proceeds and the guest program performs a system call, the *OS call emulator* translates the OS call into an appropriate call (or calls) to the host OS and then handles any associated information returned as a result of the call.
- The runtime must also handle traps that may occur as a result of executing either an interpreter instruction or a translated instruction, and it must handle any interrupts that are directed at the guest process. The runtime does this with the

exception emulator. In some cases, the exception emulator takes over when an emulation-triggered trap causes an OS signal to be delivered to the runtime; in other cases, emulation routines discover an exception condition and jump to the exception emulator. An important aspect of runtime exception handling is generating the correct, precise guest state (including program counter, register values, and trap conditions) when an exception condition occurs.

- *Side tables,* i.e., data structures generated as part of the translation process, are used by the runtime as part of the overall emulation process. One important use of side tables is in implementing a precise exception model with respect to the source ISA. Other uses of side tables will be described as they are introduced.

The design of all the major blocks will be described in more detail in the following sections; first, however, we define a framework for discussing and reasoning about compatibility in process VMs.

3.2 Compatibility

In any virtual machine implementation, a critical issue is *compatibility,* that is, the accuracy with which a guest's behavior is emulated on the host platform, as compared with its behavior on its native platform. Ideally, the behavior should be identical, although, as will become apparent, there are many practical situations where complete compatibility is not necessary for a process virtual machine to be useful.

When we define compatibility, we include all types of behaviors, with one important exception: We exclude differences that are purely performance related. That is, compatibility is a matter of correct functioning, not how fast the function is performed. Although performance is not part of the *definition* of compatibility, it is certainly an important *consideration* when implementing many VMs. In fact, the level of compatibility achieved sometimes depends very much on the degree of performance degradation that one is willing to accept.

In this section, we focus on VM compatibility in general and process-

level (ABI) compatibility in particular. Because of the focus on process compatibility, most of our discussion and many of the examples are in terms of process compatibility and program emulation.

3.2.1 Levels of Compatibility

One could take a purist's view and argue that compatibility requires 100% accuracy for all programs all of the time. For some system VMs, e.g., codesigned VMs, strict ISA compatibility is usually a requirement. However, such a strict definition of compatibility would exclude many useful process virtual machine implementations. Consequently, we further refine the notion of compatibility to allow for qualified forms of compatibility.

We refer to the strict form of compatibility as *intrinsic compatibility*. Another term sometimes applied to a VM with intrinsic compatibility is *complete transparency* (**Bruening 2004**). Satisfying intrinsic compatibility is based entirely on properties of the virtual machine. Intrinsic compatibility holds for all guest software and for all possible input data. This includes assembly language programs, written by a devious programmer with the intent of "breaking" compatibility. When using an intrinsically compatible virtual machine, the user is assured that any software will behave the same way as on its native platform, with no further verification being required; that is, the verification is done at the time the VM is constructed. This is typically the standard used by hardware designers for ISA compatibility in microprocessors. However, it is a very strict requirement for a process VM; in many cases it is more strict than is really necessary or than can be achieved.

Probably more useful for process VMs is what we call *extrinsic compatibility*. This form of compatibility relies not only on the VM implementation but also on externally provided assurances or certification regarding properties of the guest software. Extrinsic compatibility may hold for some programs run on a virtual machine but not

for others. For example, all programs compiled with a certain compiler and that use a certain set of libraries may define a level of extrinsic compatibility. Alternatively, a program may be compatible as long as it has limited resource requirements; e.g., its architected memory space requirements are less than the maximum that can be supported on a native platform. Or a software developer may take a particular program and, through debugging and other verification techniques, declare or certify that the program will give compatible behavior when run on a given process VM. For example, the Digital FX!32 system was developed with the goal of providing transparent operation; however, Digital maintained a list of Windows applications that were certified to be compatible. With extrinsic compatibility comes the requirement of clearly stating the external properties that must hold in order for compatibility to be achieved, e.g., the compiler that should be used or the logical resources needed.

3.2.2 A Compatibility Framework

As a practical matter, proving that compatibility holds, whether intrinsic or extrinsic, is an extremely difficult problem. In systems as complex as virtual machines, compatibility is usually ensured via test suites along with the logical reasoning of people intimately familiar with the system at hand. Because compatibility is too difficult to prove in any rigorous way, we would at least like some framework within which we can reason about compatibility and discuss compatibility issues.

Rather than consider the system as a whole, including the guest software, virtual machine software, operating system, and hardware platform, we first decompose the system and structure it in a way that is consistent with the way process VMs are usually constructed. Then we can reason about the parts.

To build a framework for discussing compatibility, we consider the isomorphism described in **Chapter 1 (Figure 1.2)**. That is, we focus

isomorphism described in Chapter 1 (Figure 1.2). That is, we focus on (1) the mapping of state (or "resources" that contain the state) between the guest and the host, and (2) the operations that transform the state in the guest and the host. An application program performs operations on state in two ways: by executing instructions belonging to the user ISA and through operating system calls (made either explicitly or via traps and interrupts). Consequently, we divide the state into *user-managed state* and *OS-managed state.* The user-managed state is operated on by the user-level ISA and consists primarily of architected main memory and registers. The OS-managed state includes the contents of disk files and other storage devices as well as the state associated with resources such as graphics displays and networks.

State Mapping

With regard to the user-managed state, we generally assume straightforward mappings between guest and host states. The resource holding the state does not necessarily have to be of the same type, however. For example, guest registers may be mapped to host main memory. The key point is that for a given resource holding guest state, e.g., specific register or memory location, the associated resource in the host is readily identified, and it can be easily determined whether the associated guest and host states are equivalent. The operating system–managed state is a little more difficult to deal with because it can be held in many forms, and operations on the state are often expressed in terms of OS abstractions; however, the state-mapping concept is the same.

Operations

As a program runs on a native platform, it executes user-level instructions, occasionally transferring control to the operating system (through system calls, traps, or interrupts). When the operating system is finished servicing a call, trap, or interrupt, it transfers control back to the user-level instructions. The user-level instructions mod

back to the user-level instructions. The user-level instructions modify the user-managed state, and the operating system performs operations that modify the OS-managed state and/or the user-managed state. The points where control is transferred between the user code and the OS (i.e., traps and OS calls) are a key element of our compatibility framework.

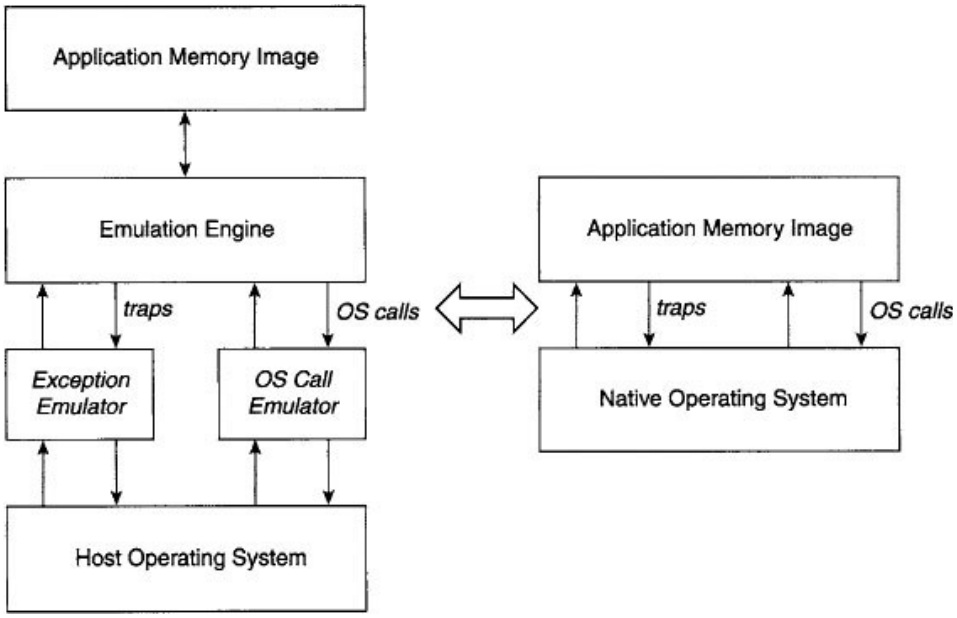


Figure 3.3 Correspondence Between Transfers of Control Between User Instructions and the Operating System in a Process Virtual Machine (left) and a Native Platform (right).

In a process VM, there are sets of operations and control transfers that correspond with those on the native platform. The emulation engine uses interpretation and/or binary translation to emulate user instructions, and the OS and exception emulators perform the operating system functions. To do this, the runtime's emulation routines may or may not call the underlying host OS, depending on the operation being emulated. In any case, for each transfer of control between user code and the OS on a native platform, we identify a corresponding point where control is transferred in the virtual machine (**Figure 3.3**). Establishing a one-to-one mapping between these control transfer points is a portion of the structure we impose on process VMs as part of our framework. In most implemented process VMs such a mapping can be established, so this requirement

is not practically limiting. Given this mapping, we can then focus on equivalence of mapped states at the points where control is transferred between user instructions and the OS (and vice versa).

Sufficient Compatibility Conditions

Given that corresponding control transfer points have been identified, we give the following conditions for compatibility.

1. At the point of control transfer from emulating user instructions to the OS, the guest state is equivalent to the host state, under the given state mapping. An important consequence of this condition is that equivalent user-managed state does *not* have to be maintained at an instruction granularity; it must only be maintained at the granularity of OS control transfers. These are the only points where the state may be made visible to the "outside world." This allows for substantial flexibility in the way that instruction emulation can be done. In particular, translated target code can be reorganized and optimized substantially —as long as the user-managed state is the same when control is transferred to the OS or exception emulator.

   On the other hand, we make the conservative assumption that *all* the user-managed state may be exposed to the outside world when the OS takes over, so at that point in time all guest and corresponding host states are required to be equivalent. For example, a system call that performs a file read or write may only modify a specific portion of the guest process's memory state, not all of it. Yet we require that all the state be equivalent. To make less conservative assumptions would require deep analysis of the underlying host OS to determine the portion of the user-managed state that may be read or written by each of the host OS system call or trap handlers. This may be straightforward in some cases, such as file I/O, but in others it may not be. To avoid analysis of the host OS, we make the conservative assumption that all the user-managed state may potentially be accessed and must therefore be equivalent at the time of control transfers.
2. At the point of control transfer back to user instructions, the guest state (both user managed and OS managed) is equivalent to host state, under the given mapping. The combined runtime and host OS actions, which emulate the guest's native OS actions, should produce system behavior, as well as modifications to the guest state (file contents, for example), that are equivalent to that for the guest's native platform. Here, an additional consideration is that for some OS behavior, e.g., driving a graphics terminal or a network interface, the ordering of operations must also be equivalent as well as the overall state changes observed

at the control transfer point.

Figure 3.4, based on the VM isomorphism of Figure 1.2, illustrates guest operation on its native platform and the corresponding operation on the host platform. On its native platform, the guest software transitions back and forth between user instructions and OS operations. At each of these transition points, a similar transition point in the host platform can be identified. At these points, the states mapped by *V* must be equivalent.

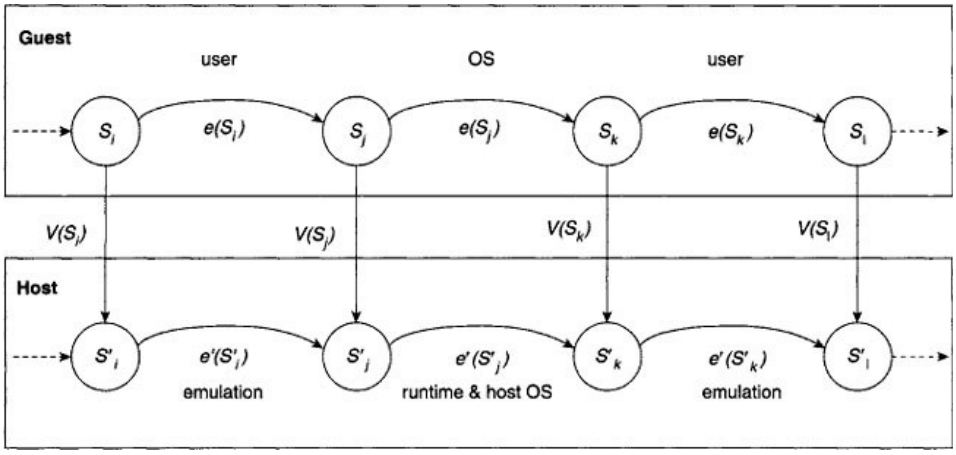


Figure 3.4 Compatibility, Illustrated in Terms of the Isomorphism Between Guest and Host States in a Process VM. *At points where control is transferred between the OS and user instructions, the states must be equivalent.*

Discussion

The compatibility framework just outlined assumes that process VMs are structured in a certain way, and the sufficient conditions for compatibility are stated relatively informally. However, the conditions are formal enough for us to discuss compatibility issues in this and later chapters in a concise manner. It should also be evident that a process VM does not have to be structured the way we assume (along the general lines of Figure 3.2), and it does not have to satisfy the foregoing conditions to be compatible. Rather, we assume a structure that is common to most realistic process VMs, and the sufficient conditions that we consider are reasonably good for compatibility, given the commonly used VM structure. The important thing is that by having such a compatibility framework, one can

more easily identify compatibility issues when they arise and then evaluate their significance.

As we have defined it, to achieve intrinsic compatibility the conditions just stated must hold for all programs and data. In contrast, for extrinsic compatibility, conditions are imposed on a subset of guest software. These conditions may be described in terms of any of the features of the compatibility framework. We now give a number of examples.

With respect to state mapping, the memory address space available to a guest program may be less than the maximum size on a native platform; consequently, a proper state mapping can only be provided for programs whose memory space requirements do not exceed a certain VM-imposed limit. Hence, extrinsic compatibility is achieved only for processes that do not exceed a certain size.

With respect to mapping of control transfers, there may be situations where some control transfers to the OS are potentially eliminated during binary translation. This can occur if certain potentially trapping instructions are removed during the target code-optimization process (as we will see in **Chapter 4**). Here extrinsic compatibility can be achieved for those programs where the trap conditions are known not to occur (possibly depending on the input data).

With respect to user-level instructions, there may be cases where guest floating-point arithmetic is not done exactly as on its native platform (see **Section 2.8.3**). For these VMs, extrinsic compatibility can only be achieved for programs where the floating-point accuracy is sufficient to satisfy the user's needs.

With respect to OS operations, there may be cases where the host OS does not support exactly the same functions as the guest's native OS. The guest may avoid these OS features as a way of achieving extrin-

sic compatibility.

Compatibility issues in process VMs receive an enlightening and thorough discussion in Breunig's thesis (**Bruening 2004**). In that work, some compatibility issues are refined further than we do here. For example, memory state equivalence of the heap and stack are considered separately.

3.2.3 Implementation Dependences

The division between architecture and implementation is central to the design of computer systems (**Section 1.2**). This division, as exemplified by the ISA, separates the functional features of a design from implementation features. Nevertheless, there are situations where implementation features do become visible in the architecture and cause functional differences. These often-subtle effects can sometimes make completely accurate emulation difficult at best.

Probably the most common examples involve caches. One example such occurs in processors that have separate instruction and data caches, as is commonly done today. If the program writes into its own instruction space (i.e., self-modifying code), some processor implementations do not automatically update the instruction cache contents (or flush the instruction cache, which would have the same effect). Rather, the old version of the modified instructions remains in the cache until it is replaced. This can lead to nondeterminism because the results of a program may then depend on external events such as context switches, which affect the instruction cache contents. To avoid this problem it is up to the programmer (or compiler) to flush the instruction cache after self-modifying code is executed. This can be done either explicitly, via instructions provided for this purpose, or implicitly, through a carefully constructed code sequence that causes the modified cache locations to be replaced. The net effect, however, is that the instruction cache becomes visible to the architecture—its presence can affect the function of software.

This might seem like a small problem. In fact, during emulation, it is sometimes easiest to make self-modifying code execute the "logical" way, i.e., by always using the most recent version of code. This can be done, for example, by using the techniques to be described in **Section 3.4.2**. However, there have been cases where clever programmers use the interactions between instruction caches and self-modifying code as a way of identifying exactly which processor implementation is executing their software and then using this information to select, at run time, a version of a procedure that is optimal for the given implementation. What if code of this type finds a "nonexistent" hardware implementation, as might be caused by emulation?

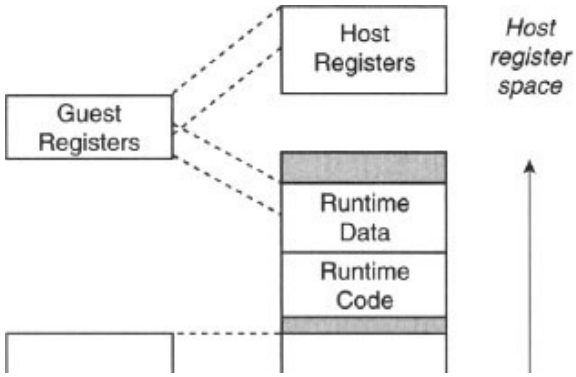
An even more troublesome scenario can arise if real implementations of an ISA provide a stricter implementation than is required by the specification. Consider the instruction cache example again. An ISA specification may state that self-modifying code is only guaranteed to take effect if the cache is explicitly flushed by executing a certain instruction; otherwise, the results are undefined (there are real ISAs specified this way). However, it may be that all existing implementations actually implement self-modifying code in the "logical" way, even when the instruction cache flush instruction is not executed. Consequently, there may be self-modifying code that does not flush the instruction cache properly yet still "works" on real implementations. Now, a developer of a VM may depend on the instruction cache flush instruction to trigger the flushing of binary translated code residing in a code cache. This not only means that self-modifying code without the explicit instruction cache flush may behave differently on the virtual machine implementation, but it may cause some guest code to "break," from the perspective of the user. In fact, the VM implements the specification and is correct, strictly speaking, but this would be of small consolation to the user whose code no longer works.

There is often no good solution to implementation “leaks” of the type just described, but it is a problem that a VM developer should be keenly aware of. Great care should be taken in reading the fine print of architecture specifications and hardware manuals that describe implementation dependences. Significant effort should be given to imagining complex scenarios where implementation dependences can become observable and may even be exploited. The bottom line is that vigilance is probably the only answer to such implementation dependences.

3.3 State Mapping

We are now ready to construct process VMs. We begin with the mapping of user-managed state, primarily state held in registers and memory. Then in later sections we discuss various aspects of emulation, i.e., the operations that transform the state.

When we describe state mapping, we often do it in terms of resource mapping, because it is the resources (registers and memory) that actually contain the state. For example, if we say that a guest memory location maps to a host memory location, we are implying that the state held in the two memory locations is the same. **Figure 3.5** illustrates a typical state mapping. In this example, the guest data and code map into the host’s user address space and share the user address space with the runtime code and data.



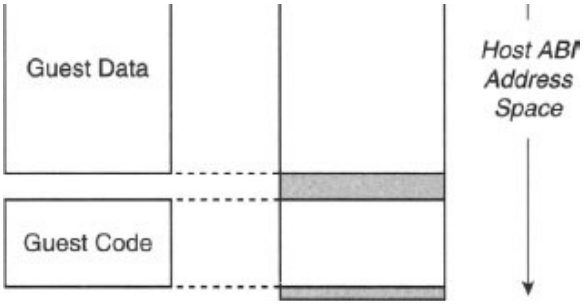


Figure 3.5 Mapping of Guest State to Host State.

Because a process VM views memory as a logical address space supported by the host machine, when we speak of *memory* in this context, we are referring to a logical address space, not the implemented real memory. This view is reflected in **Figure 3.5**, where regions of the guest memory address space map onto regions of the host memory address space.

The guest register state is mapped into host registers and/or is maintained in the runtime data region of memory. This latter mapping illustrates an important point—that the guest state does not have to be maintained in the same type of resource as on a native platform. Similarly, in some situations it may be convenient to map guest memory onto file system storage rather than the host's main memory. However, the way that these state mappings are done has a significant bearing on the emulation techniques used and on the resulting performance that is achieved.

3.3.1 Register Mapping

Register space mapping is fairly straightforward and was discussed in **Sections 2.5** and **2.8.1**; we summarize here. If the number of guest ISA registers is less than the number of host ISA registers, it may be possible to map all guest registers into host registers for the duration of emulation. In addition, there is usually a register context block maintained in memory by the runtime. The runtime loads and unloads register contents from/to this block as the emulator is entered and exited. That is, the runtime may use all or most of the

registers when it performs tasks other than emulation and then turns the registers over to the guest whenever emulation is in progress.

If the number of registers in the guest and host are the same, or nearly so, it may be theoretically possible to map all host registers to guest registers, but this could pose a problem if the emulation process requires registers for its own use (as is the case with interpretation) or if the runtime performs dynamic optimization of frequently executed code. In some ISAs, loading and unloading registers from the context block can be a problem because in order to perform a store to the register context block, a register is needed to hold the address of the context block. If all the registers are used by the emulation process, then this additional register is simply not available.

If the number of guest registers is larger than the number of host registers, then some of the guest registers must be mapped to a register context block in the host's memory. The runtime, usually through the translator, is then responsible for managing the host register space by moving guest registers into and out of the host registers as emulation proceeds.

3.3.2 Memory Address Space Mapping

The guest and host instruction sets each have their specific memory architectures, and it is the job of the runtime emulation process to map the guest's address space to the host's address space and to maintain protection requirements. Specifically, when a guest program performs a load, a store, or an instruction fetch, it is to some address *A* within the guest's address space defined by the guest ISA. However, after state mapping, the data or code at guest address *A* may actually be placed at some other address, *A′*, within the host's address space, and the runtime emulation software must be able to perform the required address space mapping. Just as with instruc-

perform the required address space mapping. Just as with instruction emulation, there is a range of possibilities for performing address space mapping spanning different performance levels, determined by the relative amount of emulation performed by software compared to the amount performed directly on the host hardware.

Runtime Software-Supported Translation Tables

Figure 3.6 illustrates the most flexible method for memory architecture emulation. As with instruction interpretation, described in the previous chapter, the more flexible methods also tend to be more software intensive. Figure 3.6 shows a software translation table maintained by the runtime, similar to a conventional page table. The guest addresses do not have to be contiguously mapped in the host address space. The guest address space can be divided into blocks, each of which, in practice, would be some whole multiple of the host page size. Then a guest address is translated to a host address by performing a table lookup to find the offset within the host space where the guest block begins.

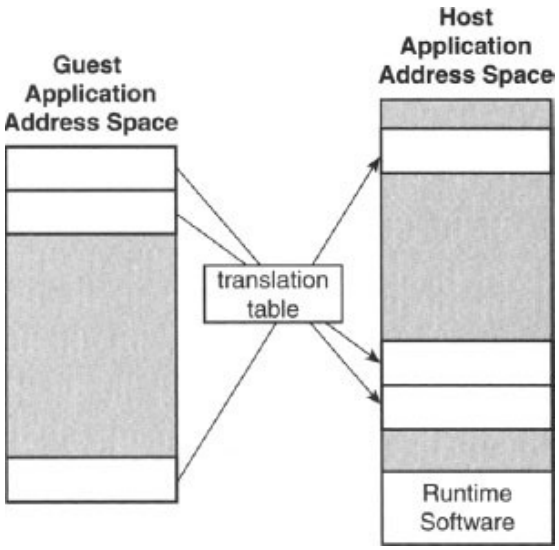


Figure 3.6 Emulating Memory Address Architecture via a Software-Implemented Address Translation Table.

```
Initially, r1 holds source address
           r30 holds base address of translation table

srwi   r29,r1,16          ;shift r1 right by an immediate 16
```

```
slwi   r29,r29,2          ;convert to a byte address in table
lwzx   r29,r29,r30        ;load block location in host memory
slwi   r28,r1,16          ;shift left/right to zero out
srwi   r28,r28,16         ;  source block number
slwi   r29,r29,16         ;shift up target block number
or     r29,r28,r29        ;form address
lwz    r2,0(r29)          ;do load
```

Figure 3.7 Code Sequence for Performing a Load Instruction with Software-Mapping Method.

Figure 3.7 contains an example of code that emulates a load instruction using a software-implemented translation table. In this example, it is assumed that the mapped memory blocks are each 64KB. The translation table itself is maintained as part of the runtime, and its base address is held in host register r30. To perform the load, the source address is shifted by 16 bits to get the 64KB block number. This number is converted to an offset added to the table base, and the sum is used to access the proper table entry. This entry contains a pointer to the address block in the host address space. The block address replaces the block address in the original guest address and can then be used for performing the requested load. Actually, the PowerPC has two instructions designed for these types of operations: the rlwimi (rotate left word immediate then mask insert) and rlwinm (rotate left word immediate and with mask) instructions. We use a longer sequence here in order to make the overall operation clearer.

The similarity to the address translation used in conventional virtual address systems for converting virtual to real addresses should be apparent. **Figure 3.8** depicts a method that extends this similarity further. Here, the runtime opens a disk file for holding data belonging to any portions of the guest address space that are not resident in the host address space. Although similar to a conventional virtual memory, the guest virtual address space is actually mapped to a region of the host virtual address space rather than to real memory. The host virtual addresses space, in turn, is mapped to the host real memory by the underlying host system.

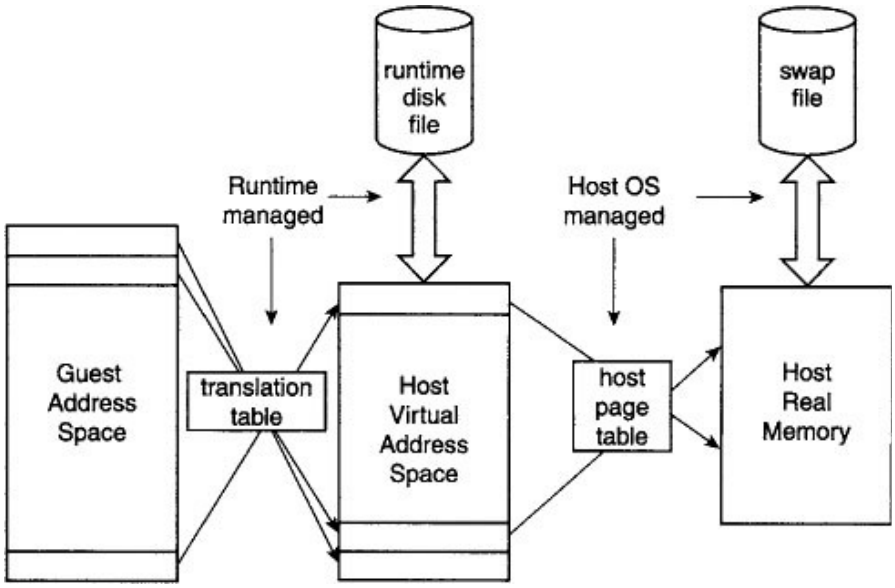


Figure 3.8 Mapping a Guest Address Space into a Region of the Host Space. *The runtime software manages the region of host address space in much the same way as the host OS manages its real memory.*

At any point in time, a given guest memory block may or may not be present in the host memory. Hence the translation table is enhanced with a "valid" bit to indicate whether or not the accessed guest block is present in the host address space. The runtime software accesses the translation table, not only to perform the address lookup but also to test the valid bit and branch to memory management software in the runtime when an accessed block is not present. The runtime memory manager then decides which block to replace, performs the necessary disk I/O, modifies the translation table, and then returns to emulation.

This step-by-step software-intensive mapping method is conceptually similar to the simple decode/dispatch interpretation for instructions. However, its use need not be restricted to interpretation; it can serve with any emulation scheme, including binary translation. The relative overhead of the scheme would be considerable in a binary translation environment but may be unavoidable if there is a semantic mismatch between the guest and host ABIs. A prime example is the case where the host application address space is not large enough to hold a typical guest application, e.g., if a 64-bit guest is be-

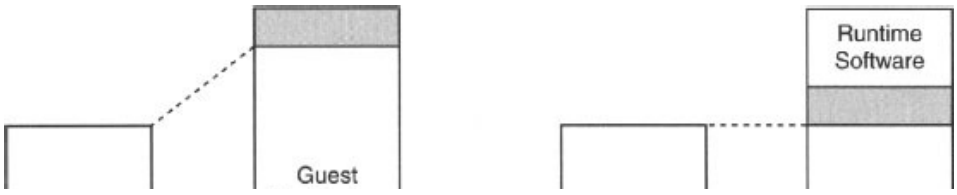
ing emulated by a 32-bit host platform. The key point is that *if all else fails,* this software-intensive method can be used.

Direct Translation Methods

We next consider address space mapping methods that rely more on the underlying hardware and less on VM software. As such, they are somewhat analogous to the binary translation methods for instruction emulation. There are two direct translation methods, as illustrated in **Figure 3.9**, one being a special case of the other.

Figure 3.9a shows a mapping where the guest address space has been displaced by a fixed offset but is still contiguous within the host space. In this case, the offset value can be held in a host register that is added to each guest address during emulation. In the example shown in the figure, the runtime is placed at the beginning of the host address space, and the guest program is placed on top, each location in the guest address space offset by a fixed amount to determine its location in the guest address space. This fixed offset mapping is used in the emulation code given in **Figure 2.16**.

Figure 3.9b shows the important special case where the fixed offset is 0, meaning that the guest addresses have the same locations in the host virtual address space. The runtime software now needs to be located in a region beyond the portion of the address space allocated for the guest, as shown in the figure. This results in a very simple mapping of guest addresses to host addresses during emulation. In this case, source memory load and store instructions can often be translated one-to-one to target loads and stores, with no need for any additional address translation instructions.



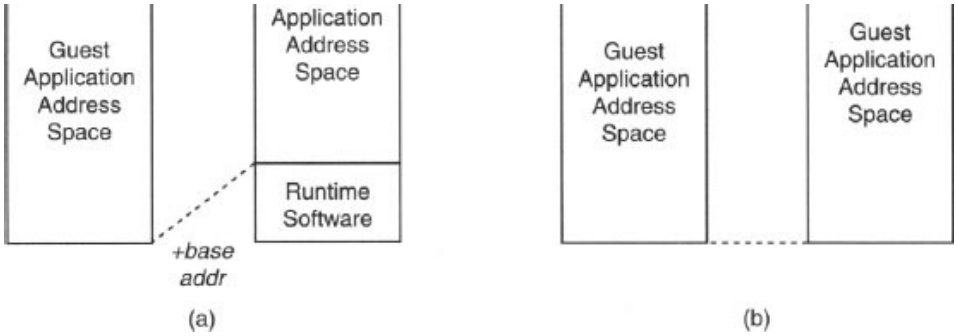


Figure 3.9 Direct Mapping of a Guest Address Space into the Host's Space. *(a) When the guest addresses are offset by some constant in the host's space; (b) when the guest addresses are placed at the same locations within the host's space.*

Compatibility Issues

Depending on the performance and compatibility level desired, the relative sizes of the guest and host address spaces have significant implications when selecting the memory mapping and translation method to use. An important consideration is that the runtime software (both code and data) share the same address space with the guest application.

If one wants intrinsic compatibility and a high-performance VM implementation, then it is probably necessary for the host address space to be larger than the guest address space so that both the maximum-size guest process and the runtime software can fit into the host address space at the same time. Then a direct mapping method can be used, as in **Figure 3.9**b. This is often the case, as when the IA-32 ISA is emulated on a platform containing a 64-bit ISA (**Baraz et al. 2003**; **Hookway and Herdeg 1997**).

If the host address space is not larger than the combination of the runtime and the maximum guest process, then either performance or intrinsic compatibility must be sacrificed. Using a software translation method as in **Figure 3.8** will yield intrinsic compatibility at the cost of several target instructions per source load or store instruction.

On the other hand, the memory actually used by a guest application

On the other hand, the memory actually used by a guest application is not necessarily the maximum size permitted by the host ABI; in most cases, in fact, a guest application does not reach its maximum size. Furthermore, in many situations users compile programs using standard libraries (of unknown size to the typical user), and most user programs do not rely on a specific memory size or on specific memory addresses. In these situations a form of extrinsic compatibility is quite sufficient. In particular, compatibility is achieved as long as the virtual memory requirements of the guest do not exceed what is left over after the runtime is placed in memory.

Furthermore, if the runtime is strategically placed with respect to the guest process, then efficient direct translation can be used. By *strategic placement,* we mean placement in a region that is not being used by the guest process. This might mean that the runtime is implemented in position-independent manner so that it can operate correctly regardless of where it is placed in memory. This allows the runtime to place itself in a region of memory that does not conflict with the locations in use by the application. Depending on loader conventions, some applications may be placed in upper memory, others in lower memory, and still others in a number of noncontiguous memory regions. In addition, the runtime can potentially be moved during the course of emulation as the application allocates or deletes blocks of memory.

3.4 Memory Architecture Emulation

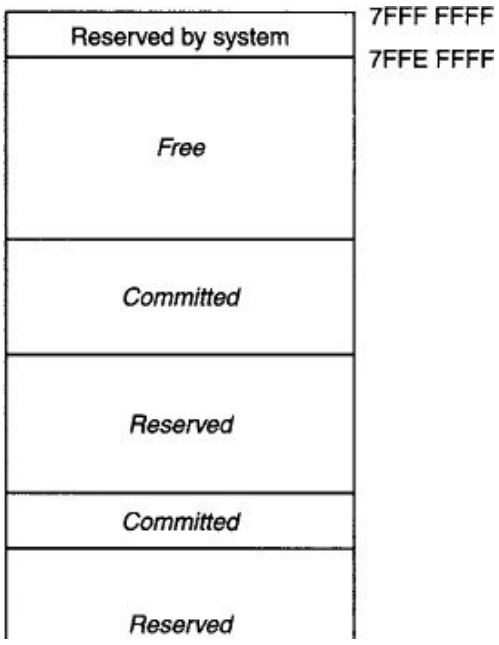
Given an address space mapping and implementation method, there are several aspects of memory architecture that must be emulated in a process VM. Specifically, there are three important features of ABI memory architecture, i.e., memory as seen by a user application, that must be taken into consideration:

- **The overall structure of the address space,** e.g., whether it is divided into segments or is a flat, linear address space. For most of our discussion, we assume a flat, linear address space because most current ABIs use a linear address space

(although it may be further subdivided into a heap and stack, for example). Techniques for virtualizing segmented memory can be built upon these linear space techniques.

- **The access privilege types that are supported.** Some ABIs support read, write, execute (R, W, E) privileges, while others restrict their support to only R and W.
- **The protection/allocation granularity,** i.e., the size of the smallest block of memory that is allocated by the operating system and the granularity at which protection privileges are maintained. In most systems the granularity of memory allocation is the same as that for protection, even though, strictly speaking, they need not be the same.

Most common ABIs define a specific size and range of addresses that are available to a user. **Figure 3.10** illustrates the address space defined for the Win32 ABI. Two 64KB blocks (at the top and bottom of the 31 bit address space) are reserved by the system. All other locations in the 31-bit address space are available to a user process. The user may reserve blocks of memory for later use. A user may also commit a region of memory, which means that the user is given access and that disk space is allocated for paging. A key point is that a user process is granted access to all addresses within the non-reserved range. As pointed out earlier, this may have important implications because the runtime must share the address space with a user process in a transparent way.



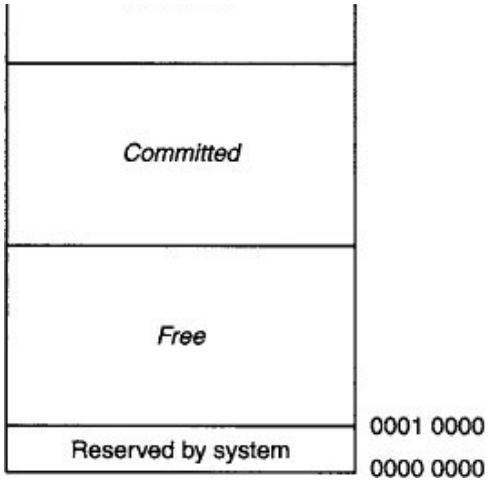


Figure 3.10 Win32 ABI Memory Architecture. *The system reserves regions at the upper and lower ends of memory. The user can reserve a specific range of addresses and later commit it to use or leave it free.*

3.4.1 Memory Protection

An important aspect of memory architecture is memory protection. Most ISAs allow access restrictions to be placed on different regions of the memory space. These restrictions are usually specified as some combination of read, write, and execute (or none).

If a software translation table is used (**Figure 3.6**), then emulation of protection checking is straightforward. Protection information can be kept in the translation table, and checking can be completely emulated as part of the translation process. This is essentially a software version of the way it is done in a conventional virtual memory system. As pointed out earlier, this method is functionally correct but slow.

With direct or offset addressing (**Figure 3.9**), however, there is no software translation table for maintaining privilege information. Clearly, a software table can be added for the special purpose of protection checking, though this method suffers in efficiency. Depending on the support provided by the host OS, a more efficient method that relies on the underlying host hardware for protection checking can be implemented. This requires that the runtime has

some means of directing the host OS to implement page protection matching the guest emulation requirements. The nature of the host support and its use in implementing efficient protection checking are described in the next subsection.

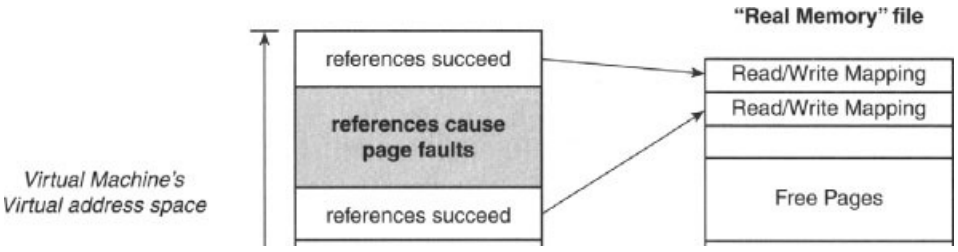
Host-Supported Memory Protection

Memory protection can be emulated by a host platform using two commonly found host OS features:

1. A system call invoked by the application (via the runtime in this case) specifying a page and the access privileges (read, write, execute, none) for the page.
2. A signal for a memory access fault that can be delivered to the runtime whenever a memory access violates the privilege specified for the containing page.

Some host operating systems support these features directly. In Linux, for example, the mprotect() system call and the SIGSEGV signal provide these functions. The mprotect system call has arguments that are (1) a virtual address aligned to a page boundary, (2) a size, and (3) a protection specification that is the bitwise OR of none, read, write, execute. The SIGSEGV signal is delivered if a page is accessed with the wrong privileges.

In addition to the direct methods supported by the OS as described earlier, there are indirect ways of emulating protection features. For example, the mmap() Linux system call can be used for mapping a region of the address space for which the runtime wants access control. The region is mapped to a file that has the required access privileges, as shown in **Figure 3.11**. If the application software attempts a disallowed access, a SIGSEGV signal is delivered (**Bovet and Cesati 2001**).



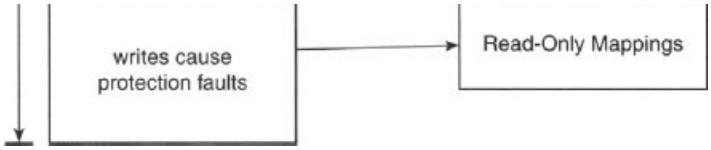


Figure 3.11 Mapping a Portion of the Guest Virtual Address Space to a File. *The mapping can be controlled via mapping, unmapping, and read-only mappings.*

In the absence of any of these convenient ways to provide the runtime with the ability to control access to the guest software through host-implemented features, the runtime can always fall back to the less efficient software mapping method of accessing memory.

Page Size Issues

We now consider the case where the host and guest pages are of different sizes; obviously, this difference must be considered when allocating and protecting memory. As far as assigning specific access privileges to the guest pages, the method described in the preceding section is fairly simple when the guest page size is a multiple of the host page size. In this case all the host pages that correspond to the single guest page are given the same protections. However, when the guest page size is smaller than that of the host, two or more different guest pages may be included in the same host page. This causes a problem if the protections are different for guest pages that share the same host page. **Figure 3.12** illustrates a case where a guest code page and data page share the same host page. The guest pages should have different access privileges, but in the host system they must be given the same protections. As always, the software-based mapping method can be used to solve this problem, but with significant overhead.

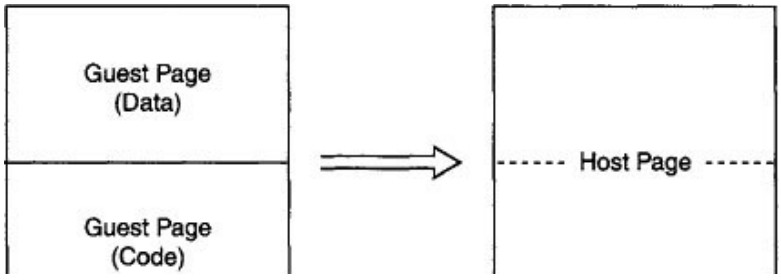




Figure 3.12 Guest Page Smaller Than Host Page. *If the guest page size is smaller than the host page size, then some host pages may be divided between code and data. This causes difficulties when emulating memory protection.*

Aligning the code and data regions on host page boundaries would seem to fix the problem, but this essentially relocates regions of the guest address space, and the ISA emulation software must correct for this realignment by adding constant offsets to guest addresses within relocated regions. This may not only affect the efficiency of emulation but may also reduce the portability of emulation code by adding a host platform dependency.

An alternative is to maintain protection checking through the host platform as described earlier, but to take a conservative approach, giving the entire host page the lesser privilege of the different guest pages that share it. The runtime must then handle any “extra” traps generated either through signal handler software or by reverting to software mapping (and privilege checking).

Another issue with matching guest and host memory architectures is that the types of protections defined in the two ISAs may not match. If the host supports a superset of guest protections, the guest can be granted the correct protections. However, if the host supports only a proper subset of the guest protections, then the VM software must use conservative protection assignments or rely on software checks, as described earlier.

An important case is where the host supports only RW privileges while the guest uses RWE. This is the most common case of a mismatch that occurs in practice. With emulation via interpretation, the interpreter code can easily check execute protection as part of its normal operation with relatively little slowdown, i.e., through a software translation table. With binary translation, execute protection for the guest code has to be checked only at the time the runtime reads the guest code and performs translation. In addition, the run-

time must be able to detect the situations where the application changes the protection, but this can be done easily as part of the OS call translation process (**Section 3.7**). When this occurs, any affected translated code must be discarded.

3.4.2 Self-Referencing and Self-Modifying Code

Occasionally, an application program either refers to itself (i.e., it reads from its code region) or attempts to modify itself by writing into the code region. This poses potential problems when binary translation is used, because the code actually executing is translated code, not the original source code. If the original source code would have read or written itself, then the translated version must produce exactly the same results. Such instances of self-referencing and self-modifying code must be accurately emulated by the runtime.

Basic Method

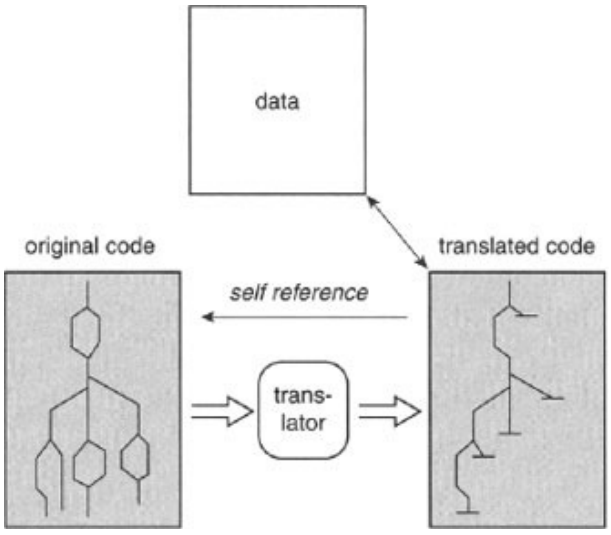
The basis for the solution is the same for both problems. In particular, an accurate memory image of the guest program code is maintained at all times. All load and store addresses in the translated version are mapped into the source memory region, regardless of whether code or data is being addressed. Consequently, the self-referencing case (**Figure 3.13**a) is automatically implemented correctly.

For self-modifying code (**Figure 3.13**b), the original source code region is write-protected by the runtime. That is, the page-level write access privilege is turned off for all pages containing translated source code. This can be done via a system call made by the runtime (e.g., via Linux mprotect()). Consequently, any attempt to write a page containing translated code results in a protection trap and the delivery of a signal to the runtime. At that point, the runtime can flush either the entire code cache or just the translations corresponding to the modified page (by using a side table that keeps track

of the page(s) from which translated code blocks originated). Then the runtime should temporarily enable writes to the code region, enter interpretation mode, and interpret forward, at least until it gets through the code block that triggered the fault. This interpretation step guarantees forward progress when a translated code block modifies itself. Then the runtime can reenable write-protection and proceed with normal operation, which will eventually result in retranslation of code in the modified page.

Pseudo-Self-Modifying Code

The method just described is a high-overhead way of dealing with selfmodifying code, but in most situations it does not hurt performance significantly because in many programs self-modifying code tends to be rather uncommon (or nonexistent). Nevertheless, there are certain programs or types of programs where self-modifying code or pseudo-self-modifying code does occur frequently enough to cause a large performance loss. *Pseudo-self-modifying* code describes the situation where writeable data areas are intermixed with code. That is, a write into a “code” page does not literally modify code, but will trigger a write-protection fault. This code sometimes appears in assembly language routines, such as device drivers, some performance-critical game kernels, and embedded code.



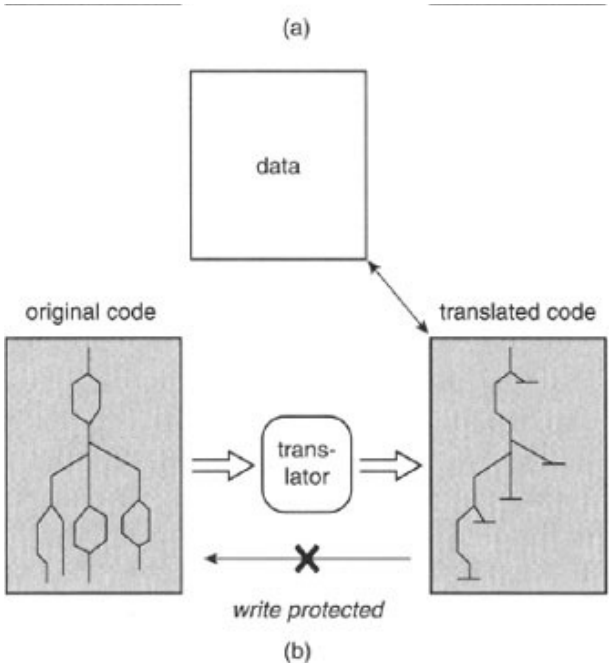


Figure 3.13 Self-Referencing and Self-Modifying Code. *Keeping a copy of the original code can be used for solving the problem of (a) self-referencing code and (b) self-modifying code.*

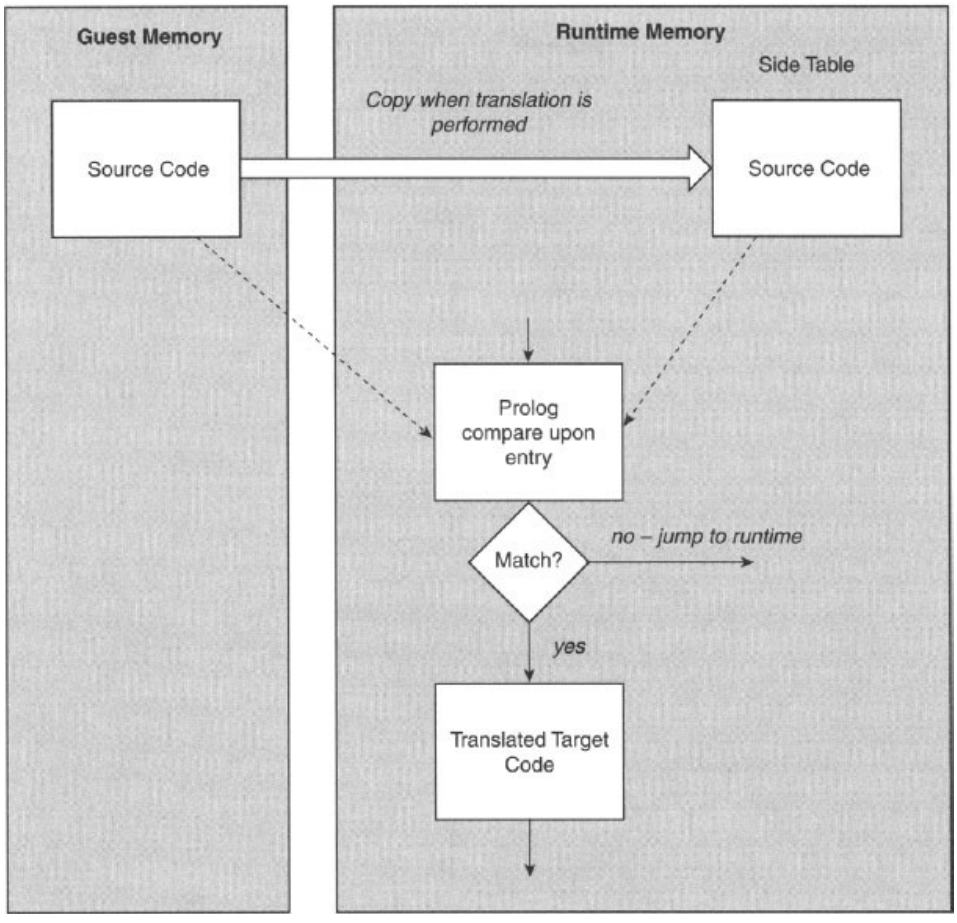


Figure 3.14 Handling Pseudo-Self-Modifying Code via Dynamic Checks

Figure 3.14 Handling Pseudo-Self-Modifying Code via Dynamic Checks.

To deal with frequently occurring pseudo-self-modifying code, one approach is to check dynamically whether the source binary code has been modified before the corresponding translated target code is executed (**Dehnert et al. 2003**). That is, if write-protection faults to the same code region repeatedly occur, the runtime can retranslate the code in the page being written to and, for each translation block, save the original source code in a side table (see **Figure 3.14**). Then as a prolog to each of the translation blocks, the translator inserts check code that compares the current version of the source code with the original side table version of the source code. Write-protection for the page in question can then be turned off. As long as the prolog check code finds the source code unchanged (and it will in the case of pseudo-self-modifying code), then execution of the translated code can proceed. This method results in slowdowns (by at least a factor of 2, because the code comparison is time consuming), but it is still much faster than repeated protection faults, interpretations, and retranslations.

A potentially faster alternative is to make the prolog code a separate code block that can be linked and unlinked from its corresponding translation block (**Dehnert et al. 2003**). Immediately after a write-protect fault, the runtime links in the prolog code, and write-protection for the modified page is turned off. The check code is executed the next time the potentially modified code is entered, and, if the source is found to be unmodified, the prolog can be unlinked and write-protection reenabled. With this method, the check is performed only the first time that potentially modified code is executed following a write-protect fault. However, there is problem in making forward progress when the code that performs the write is in the same page that is being modified. That is, the code repeatedly traps, disables write-protection, later finds that the source code region is unmodified, reenables write-protection, and then modifies the source code region, resulting in another trap. The probability

that this will happen can be reduced if source code can be write-protected at finer granularity than a page, as described in the next subsection. In any case, the slower method given in the foregoing paragraph can always be used as a fall-back.

Fine-Grain Write-Protection

The basic write-protect method, as we have been considering it thus far, protects the source code region at a page granularity; if there is a write into the page, all the translations formed from instructions in that source code page are flushed from the code cache. An improvement on this approach uses software to keep track of source code blocks at a finer granularity (**Dehnert et al. 2003**). For each page that contains translated source instructions, the runtime can maintain a finer-granularity protection table, containing a write-protect bit mask per page. Each bit in a page's mask corresponds to a small region of the page, e.g., 128 bytes. Then as code is translated, the translator sets bits in the mask as it translates source instructions. If a particular page contains a combination of code and data, then any data-only regions (at the 128-byte granularity) will not have their fine-grain write-protect bits set. When a write into a source code page results in a trap to the runtime, the runtime can retrieve the fine-grain write-protect mask for the given page and compare the faulting address with the bits in the bit mask. If, for example, the faulting write was to a data-only region, then the write-protect bit for that region will not be set, and translated instructions do not have to be flushed. The finer-granularity bit mask can also be used to reduce the flushed translations when a fine-grain code region is written to (through the use of side tables that track the translation blocks corresponding to instructions in each of the fine-grain regions).

True Self-Modifying Code

For cases where true self-modifying code frequently occurs, an ap-

proach for avoiding write-protect faults for the binary translator is to remove the self-modifying code via idiom recognition. That is, the translator code recognizes the common scenarios where self-modifying code occurs and invokes special translations. For example, self-modifying code may be used for writing the immediate field of an instruction contained in an inner loop and then executing the loop. This is illustrated in **Figure 3.15**a, where the immediate field of an IA-32 add immediate instruction is frequently modified. This case can be handled by converting it into equivalent data region read/write operations. In **Figure 3.15**b, the add immediate instruction is translated to a load that retrieves the immediate field from the source code (which is a data region from the perspective of the translated code) followed by an add instruction that uses the register value rather than an immediate.

In other situations, a given region of source code may be modified in only a small number of different ways. This occurs in some device-independent drivers, where the driver itself writes an implementation-dependent piece of code and then executes it. In these situations, the runtime can save the various source code versions along with the corresponding translated code. Then when code modification is detected, the runtime can compare the modified source code against the saved source code versions. If it finds a match, the corresponding saved translation can be retrieved and used. This method still incurs significant overhead, but at least it avoids repeated retranslations.

```
label:  add %eax, 0x123456          ;add immediate
```

(a)

```
        lwz r29, label+1(r2)        ;r2 points to the IA32 memory image
        add r4,r4,r29               ;r4 contains %eax
```

(b)

Figure 3.15 Translating an Instruction Containing a Frequently Modified Immediate Field. *(a) An IA-32 add immediate instruction; (b) a PowerPC translation, where the immediate value is loaded from the original source code memory image.*

Protecting Runtime Memory

Because the runtime software (including both its code and data) shares an address space with guest application software, the runtime must be protected from the guest application. For example, if an emulated program should, through a bug or by intent, attempt to access a region in the address space where the runtime resides, then this should be reported as a memory fault with respect to the emulated application. To maintain compatibility and give correct results, the guest program should not be allowed to read from or write into the runtime's memory area and continue on its way.

If a software translation table with protection checking is implemented (**Figure 3.6**), then such an access violation can easily be detected as part of the translation process. The runtime maintains access privileges in the memory map, so the runtime itself is protected. Although effective, this method would be quite slow.

A higher-performance solution is used in the Omniware VM (**Lucco, Sharp, and Wahbe 1995**; **Wahbe et al. 1993**). In this system, underlying host hardware is used for address translation (as in **Figure 3.9**), but protection checking is done in software. To streamline software protection checking, guest data and code are divided into power-of-2-sized segments. By relying on such segments, checking can be done fairly efficiently via a single shift (to extract the segment address bits) and a comparison. If only one data segment is active at any given time, then the address bits of the currently accessible segment can be held in a host register, making the comparison process even faster. Furthermore, by analyzing program control flow, optimizations similar to the ones used in HLL VMs for reducing null pointer and array range checks (**Section 6.6.2**) can be applied. This reduces the overhead even more, resulting in a reported total overhead of about 10% (sometimes less). Although this technique protects the runtime from out-of-bounds accesses made by the guest, forcing power-of-2 segments may compromise intrinsic