10.4 Virtual Machines

An instruction set architecture implemented in software is known as a *virtual machine*. Programs executed by a virtual machine will experience only minor slowdowns if most of the instructions executed are among those implemented by the underlying hardware rather than by software simulation. So there is good reason for a virtual machine's instruction set to include instructions implemented by the underlying hardware. In addition, with such an instruction set, existing programs written for the underlying hardware do not have to be modified or even recompiled for execution by a virtual machine.

A *virtual machine manager* (VMM), also known as a *hypervisor*, is a software layer that time multiplexes a (hardware or virtual) processor to create one or more virtual machines. The VMM enforces isolation that protects memory, processor state, and input/output devices associated with each virtual machine from actions by other virtual machines, and it also enforces isolation to protect the VMM itself from actions by virtual machines. Whereas operating system primitives and shared input/output device allow processes to communicate and synchronize with each other, VMMs typically offer no means for one virtual machine to communicate directly with another. Virtual machine isolation thus is a stronger guarantee than process isolation.

VMMs are popular, in part, because they offer strong isolation guarantees in a form that is useful across a broad range of settings.

- Cloud providers employ VMMs to give each customer an illusion of being assigned sole tenancy on some computers. The customer then can select an operating system and entire software stack to be loaded and run on each of those computers.[14]

- In enterprise datacenters, VMMs enable *server consolidation*, whereby a single computer runs multiple virtual machines that each hosts a server. Less effort is involved in managing the one computer (with only a single console and associated set of peripherals) than would be required if each server were run on separate hardware.

- On the desktop, running a VMM compensates for weak operating system security if distinct virtual machines are assigned different tasks (e.g., one virtual machine for personal banking, one for working on your job from home, and one for casual web-browsing). The isolation of virtual machines here limits the potential impact of a compromised virtual machine and also impedes the spread of malware.

Software monitoring and debugging also is facilitated when that software is executed by a virtual machine. A typical VMM provides a *virtual console* for

[14]This kind of cloud computing is know as *infrastructure as a service* (IaaS). With *platform as a service* (PaaS), the customer is offered a computer that runs some pre-configured software stack. And *software as a service* (SaaS) connects customers with specific applications and/or databases that run in a datacenter.

each virtual machine it implements. The virtual console allows an executing virtual machine to be paused and allows that virtual machine's memory and (virtual) processor state to be inspected and/or changed—a powerful debugging tool. Application software, an operating system, or even a VMM itself now can be debugged simply by running that software in some virtual machine and using the associated virtual console.

10.4.1 A VMM Implementation

We next sketch a VMM for the hypothetical CPU introduced above in §10.2.2. In this implementation:

- Address translation is employed by the VMM to ensure that no virtual machine can retrieve or alter memory allocated to the VMM or to other virtual machines.
- Time multiplexing is employed by the VMM to ensure that no virtual machine can retrieve or alter the processor state of others.

So no virtual machine should be able to change the `MmapReg` or `IntVector` registers. User-mode instructions cannot change the `MmapReg` or `IntVector` registers, which suggests a design where virtual machines—whether in (virtual) user or system mode—run on a processor that is in user mode. The VMM then would execute in system-mode; it intercepts and simulates interrupts and system-mode execution for each virtual machine.

Memory Isolation. Our VMM has a separate memory map $VMap_V$ for the memory of each virtual machine V. These memory maps (by design) will have disjoint ranges, thereby relocating the memory of different virtual machines to non-overlapping memory regions in the underlying processor. In addition, $VMap_V$ is defined in a way that blocks acccesses by V to memory that is allocated to the VMM.

Software executing in a virtual machine V might itself install a mapping *Mmap* by loading *Mmap* into the V's (virtual) `MmapReg` register. To ensure that execution in a virtual machine is like execution on the bare hardware, the memory map used for a program executing in a virtual machine V relocates memory accesses according to both $VMap_V$ and *Mmap*— an access by V to address n is relocated to memory location $VMap_V(Mmap(n))$.

Two functions will be convenient in connection[15] with defining the combined memory maps that a VMM constructs and manipulates:

[15]We adopt a standard convention from programming languages. Identifiers on the left-hand side of an assignment statement are interpreted as specifying addresses; identifiers on the right-hand side of an assignment statement are interpreted as the specifying the value stored at the indicated address. So, for example, $MapApply(Mmap, n) := 23$ changes the value stored at address $Mmap(n)$ to 23; $y := MapApply(Mmap, n)$ stores into y the value in memory location $Mmap(n)$.

```
var  $VMTable[1 .. NumVMs]$: $virtMach$
     $LastRun$: $1 .. NumVMs$
type $virtMach$ = record
                  $ps$: processor state
                  $VMode$: $\{S, U\}$
                  $VMap$: memory map
                  $VIntPend[1 .. NumIntClass]$ : list of processor state
                  $nxtIT$: integer
                  end record
```

Figure 10.4: VMM State for Virtual Machines

$MapComb(M', M'')$ is the mapping that is the composition of mappings M and M'. After executing $M := MapComb(VMap_V, Mmap)$, we would have $M(n) = VMap_V(Mmap(n))$ for all n.

$MapApply(M, n)$ is an identifier for the memory location to which mapping M relocates n.

Therefore, to resume execution of a virtual machine V, it suffices for the VMM to load the processor's `MmapReg` register with $MapComb(VMap_V, Mmap)$ if V's `MmapReg` register had last been loaded with memory map $Mmap$.

Time Multiplexing. Our VMM uses entry $VMTable[V]$ (see Figure 10.4) to store processor state and other information associated with a virtual machine V. Interrupts and system-mode execution in virtual machines are efficiently simulated if the VMM is given exclusive control over the subset Reg_S of registers that determine processor mode, memory mappings, and interrupt handling. For our hypothetical CPU, Reg_S includes `mode`, `MmapReg`, `IntVector`, `Enabled`, and the interval timer.

Exclusive VMM control of registers in Reg_S is easily achieved if (i) any instruction for accessing these registers is system-mode (on our hypothetical CPU they are) and (ii) the processor is in user-mode whenever a virtual machine is being executed (something we already assume). Conditions (i) and (ii) suffice because they ensure that a virtual machine's attempt to update any register in Reg_S will cause a trap (which transfers control to a VMM interrupt handler).

The following invariant characterizes where the current processor state of each virtual machine V can be found:

- While V is not executing instructions on the underlying processor, its current processor state is stored in $VMTable[V].ps$.
- While V is executing instructions on the underlying processor:
  - $VMTable[V].ps.r$ contains the current value of each register $r \in Reg_S$.
  - Register r on the underlying processor contains the current value of each register $r \notin Reg_S$.

```
$Hndlr_i$: **procedure**
    **let** $vm = VMTable[LastRun]$
        $MReal = vm.VMap$
    **in**
      **for** $r \notin Reg_S$ **do** $vm.ps.r := \texttt{IntVector}[i].old.r$ **end**
      **enq**$(vm.VIntPend[i],\ vm.ps)$
      **if** $vm.ps.\texttt{Enabled}[i] = true$ **then**
        $vm.ps := MapApply(MReal, vm.ps.\texttt{IntVector}[i].new)$
        $MapApply(MReal, vm.ps.\texttt{IntVector}[i].old) :=$ **deq**$(vm.VIntPend[i]))$
      **call** $Dispatcher$
    **end** $Hndlr_i$
```

Figure 10.5: Sketch of VMM Generic Interrupt Handler

To preserve this invariant, the value in each register $r \notin Reg_S$ must be copied by the VMM to $VMTable[LastRun].ps$ whenever an interrupt causes virtual machine *LastRun* to relinquish control of the underlying processor:

$$\textbf{for}\ r \notin Reg_S\ \textbf{do}\ VMTable[LastRun].ps.r\ :=\ Intrpt[i].old.r\ \textbf{end} \qquad (10.3)$$

Therefore, this code appears at the start of every VMM interrupt handler. Preserving the invariant also requires that a VMM's *Dispatcher*, which is invoked at the end of every interrupt hander and resumes some previously executing virtual machine V, loads registers $r \notin Reg_S$ using values in $VMTable[V].ps.r$.

Virtual Machine Interrupts. The response to an interrupt of class Int_i depends on the processor's `Enabled` register. If `Enabled`$[i]$ equals *true* (so interrupts of class Int_i are enabled) then the current processor state is saved and a new processor state is loaded; otherwise, execution continues and the interrupt is queued for later delivery. The VMM simulates this behavior in its interrupt handlers.

Figure 10.5 outlines a generic VMM handler for class Int_i interrupts. The handler first updates $VMTable[LastRun].ps$, using (10.3) with the state saved by the underlying processor when interrupted. Next, the handler records the pending interrupt by appending[16] that state to $vm.VIntPend[i]$—a queue of class Int_i interrupts awaiting delivery on virtual machine *LastRun*. Finally, if class Int_i interrupts are enabled on virtual machine *LastRun* then a new processor state is loaded from its $Intrpt[i].new$, and the interrupted processor state is removed[17] from the head of queue $vm.VIntPend[i]$ and copied into $Intrpt[i].old$. Virtual machine *LastRun*'s `IntVector`$[i]$ register identifies the address of regions $Intrpt[i].new$ and $Intrpt[i].old$.

[16]Operation **enq** appends the value specified by its second argument onto the queue specified in its first argument.

[17]Operation **deq** performs a dequeue operation on the queue specified in its argument.

System-Mode Instructions. A straightforward VMM simulatation of system-mode instructions is possible provided they satisfy:

> **System-Mode Instruction Restriction.** The underlying processor signals a trap—here called a *privilege trap*—whenever execution of a system-mode instruction is attempted while `mode` = U holds. □

Now, a VMM-provided handler for privilege traps can be used to simulate execution of an attempted system-mode instruction.

Figure 10.6 sketches that handler. $Hndlr_{priv}$ does not contain code to queue pending interrupts, whereas the generic interrupt handler of Figure 10.5 does. Such a queue in $Hndlr_{priv}$ would store at most one element, because traps are always enabled, a privilege trap cannot be signalled by a virtual machine that is not executing, and no virtual machine can be executing while $Hndlr_{priv}$ is. For brevity, Figure 10.6 gives code to simulate only a few of the virtual machine's system-mode instructions: `loadpc` to load the program counter (`pc`), `loadintrpt` to load interrupt vector registers `IntVector`[i] for all i, `loadmmap` to change the `MmapReg` register, and `loadtmr` to load the interval timer. Figure 10.6 does include code (toward the end of $Hndlr_{priv}$) to simulate the privilege trap that should occur when a virtual machine *LastRun* attempts to execute a system-mode instruction while in (virtual) user mode.

Interval Timers. Each virtual machine has its own interval timer. The VMM simulates these by using two registers from the underlying processor: its interval timer and a register `TimeNow` that maintains the current time.[18] The VMM simulation of a virtual machine V's interval timer works as follows.

- *VMTable*[V].*nxtIT* stores the time when the next interval timer interrupt should be signalled on virtual machine V; a constant *MaxTime*, larger than any value ever found in `TimeNow`, indicates that no interval timer interrupt currently is scheduled.
- *Dispatcher*, prior to resuming execution of any virtual machine, invokes *SimTimers* (Figure 10.7). *SimTimers* simulates the occurence of an interval timer interrupt at every virtual machines where sufficient time has elapsed since that virtual machine's interval timer was loaded.

Note that, with this simulation, delivery of an interval timer interrupt at a virtual machine could be delayed by other activity.

Dispatcher. Each virtual machine is executed periodically, for a short time slice, on the underlying processor. The time slice begins when *Dispatcher* selects and resumes execution of a virtual machine; the time slice ends when an

[18]Most hardware processors have a register like `TimeNow`. But if such a register is not available then it can be simulated by a variable *TimeOfDay* maintained by the VMM. The VMM records in another variable *LastIT* the value it last loaded into the interval timer. And whenever the VMMs handler for timer interrupts is invoked, *LastIT* is added to *TimeOfDay*.

```
$Hndlr_{priv}$: **procedure**
    **let** $vm = VMTable[LastRun]$
        $MReal = vm.VMap$
        $MVirt = MapComb(MReal, vm.ps.\texttt{MmapReg})$
    **in**
      **for** $r \notin Reg_S$ **do** $vm.ps.r := \texttt{IntVector}[priv].old.r$ **end**
      **if** $vm.ps.\texttt{mode} = S$
        **then** {simulate system-mode instruction}
          **case** $MapApply(MVirt, vm.ps.\texttt{pc})$ {instruction being simulated}
              **when** loadpc $val$: {load program counter with $val$}
                  $vm.ps.\texttt{pc} := val$
                  **call** $Dispatcher$
                  **end when**
              **when** loadintrpt $val$: {load IntVector[$i$] with $val[i]$ for all $i$}
                  $vm.ps.\texttt{IntVector} := val$
                  **call** $Dispatcher$
                  **end when**
              **when** loadmmap $MMap$: {load MmapReg with $MMap$}
                  $vm.ps.\texttt{MmapReg} := MMap$
                  **call** $Dispatcher$
                  **end when**
              **when** loadtmr $val$: {load interval timer with $val$}
                  $vm.nxtIT := \texttt{TimeNow} + val$
                  **call** $Dispatcher$
                  **end when**
                ⋮
          **end case**
        **else** {simulate privilege trap at virtual machine $LastRun$}
              $vm.ps := MapApply(MReal, vm.ps.\texttt{IntVector}[i].new)$
              $MapApply(MReal, vm.ps.\texttt{IntVector}[i].old) := vm.ps$
      **call** $Dispatcher$
    **end** $Hndlr_{priv}$
```

Figure 10.6: Code for Privilege Interrupt Handler

```
SimTimers: procedure
    for $v \in 1 .. NumVMs$ do
        let $vm = VMTable[v]$
            $MReal = vm.VMap$
        in
          if $vm.nxtIT \le \texttt{TimeNow} \wedge vm.ps.\texttt{Enabled}[IT] = true$ then
              $vm.nxtIT := MaxTime$
              $MapApply(MReal, vm.ps.\texttt{IntVector}[IT].old) := vm.ps$
              $vm.ps := MapApply(MReal, vm.ps.\texttt{IntVector}[IT].new)$
    end SimTimers
```

Figure 10.7: VMM Simulation for Interval Timers

interrupt is signalled, so the VMM again gets control. To guarantee that such interrupts will occur, *Dispatcher* loads the interval timer on the underlying processor just prior to resuming a virtual machine.

To resume execution of a given virtual machine V, *Dispatcher* loads values, as follows, into the underlying processor's registers.

register	value loaded for resuming virtual machine V
$r \notin Reg_S$	$VMTable[V].ps.r$
`mode`	U
`MmapReg`	$MapComb(VMTable[V].VMap, VMTable[V].ps.\texttt{MmapReg})$
`IntVector`	addresses for VMM's interrupt handlers
`Enabled`	$\texttt{Enabled}[i] = true$ for all i, so all interrupts enabled
interval timer	$\min(\tau, \min_i(VMTable[i].nxtIT - \texttt{TimeNow}))$

Here is the rationale:

- The values being loaded into a processor register $r \notin Reg_S$ are prescribed by the invariant given earlier (page 283) for the processor state of a virtual machine V.

- By loading `mode` with U, a register $r \in Reg_S$ cannot be changed by V's execution. V is then prevented from changing the interrupt handlers or address translation that VMM installs.

- `MmapReg` is loaded with a composition of mappings $VMTable[V]].VMap$ and $VMTable[V].ps.\texttt{MmapReg}$. Consequently, V cannot access memory used by other virtual machines (because $VMTable[V].VMap$ is part of the composition). Also, addresses in the memory V can access are translated according to whatever memory mapping was last loaded into `MmapReg` by software executing on V (because $VMTable[V].ps.\texttt{MmapReg}$ is part of the composition).

- The values *Dispatcher* loads into `IntVector` and `Enabled` ensure that a VMM-installed interrupt handler receives control whenever an interrupt is signalled by the underlying processor. An execution of that interrupt handler, in turn, might change $\mathit{VMTable}[V]$ to cause subsequent execution by the interrupted virtual machine V to resume with an interrupt handler that V installed.

- The value that VMM loads into the interval timer bounds the elapsed time until some VMM interrupt handler next executes. That value is calculated, as follows. The length of a time slice for uninterrupted virtual machine execution is τ; the time until the next interval timer interrupt is scheduled to occur at a virtual machine V is $\mathit{VMTable}[V].\mathit{nxtIT}-\texttt{TimeNow}$. So *Dispatcher* loads the interval timer with the minimum of these, for all virtual machines.

Input/Output. Each virtual machine is provisioned with it's own complement of virtual input/output devices. Virtual input/output devices range from slower or lower-capacity variants of existing hardware to devices having new functionality and/or new interfaces. Either way, the operating system in a virtual machine would include a driver for each virtual input/output device that it supports. Virtual input/output devices that closely resemble existing real devices often don't require new drivers to be written; new devices do.

Initiation of Input/Output Operations. With *direct I/O*, the processor provides a system-mode `startIO` (say) instruction that, when executed, initiates an input/output operation; operands give details of the operation and are stored in registers and/or memory. The alternative is *memory-mapped I/O*, where each input/output device is associated with a few co-opted memory addresses (that no longer access storage). Here, an input/output operation is initiated by writing to the addresses associated with the device; the values written to those associated addresses specify details of the operation.

On most processors, unmapped (not virtual) memory addresses must be communicated to input/output devices. So system software must mediate transfers between an input/output device and a region of virtual memory. To effect the transfer, a collection *input/output buffers* are allocated and system software executes with a memory map *Mmap* where

- every memory address L in the scope of an input/output buffer satisfies $\mathit{Mmap}(L) = L$.[19]

- $\mathit{Mmap}(L) = \mathit{Mmap}_P(L)$ holds for L a virtual address in the region of Mmap_P that is the source/destination for the input/output operation.

Input/output to/from virtual memory involves two steps: (i) the input/output device transfers data to/from a designated input/output buffer, and (ii) system software copies (using ordinary load and store instructions but subject to

[19]In fact, it is not unusual to have $\mathit{Mmap}(L) = L$ for all locations used by the system software rather than just memory being used as input/output buffers.

address translation using *Mmap*) between that input/output buffer and the source/destination in virtual memory.

A VMM supports input/output device by intercepting input/output instructions—`startIO` or memory-mapped I/O—that virtual machines execute.

- Because virtual machines always executes in user-mode, a privilege interrupt will be signalled whenever a virtual machine executes `startIO`. So the VMM's privilege interrupt handler will be invoked in response to the `startIO`.
- By excluding memory-mapped I/O addresses from the domain of the memory map (*VMap*) of every virtual machine, any virtual machine's access to memory-mapped locations will signal an address-translation interrupt. So the VMM's address translation interrupt handler is invoked.

Once the VMM gets control, it executes code to perform the input/output operation being initiated by the virtual machine. That VMM code is likely to perform input/output operations on I/O devices connected to the underlying processor. Drivers in the VMM initiate these operations.

Termination of Input/Output Operations. Whenever an input/output operation terminates, an *I/O interrupt* is signalled; information about the operation's success are conveyed in specified registers or in pre-defined memory locations. The interrupt causes an I/O interrupt handler to be invoked. For some input/output devices, operations do not necessarily terminate in the order they are initiated; here, information conveyed with the interrupt will identify which input/output has terminated.

When a VMM is present then the VMM's input/output interrupt handler will receives control upon completion of an input/output operation. The handler notifies the appropriate VMM driver, which then simulates an input/output interrupt in the virtual machine that initiated the virtual input/output operation in the first place.

Example of Input/Output with a VMM. To illustrate, we sketch a VMM implementation for a virtual disk *VD*. It employs a mapping $BlkMap_{VD}$ from the blocks comprising *VD* to blocks on some set of underlying disks that the VMM controls.[20] The code for emulating an input/output operation *op* to block *b* on *VD* works by issuing *op* to block $BlkMap_{VD}(b)$ on some underlying disk. When *op* terminates, the corresponding I/O interrupt causes the VMM's I/O interrupt handler to run. That interrupt handler determines which virtual machine to notify by using information conveyed with the interrupt in conjunction with information recorded by the VMM when it first received virtual input/output request for block *b* on *VD*. The appropriate virtual machine is then notified by simulating an I/O interrupt for it—code for such a simulation would be an instance of Figure 10.5.

[20]Isolation of different virtual disks follows if each block in any virtual disk is mapped to a unique block on one of the underlying disks.

10.4.2 Binary Rewriting

The VMM described above requires a processor in which executing a system-mode instruction causes a privilege trap when `mode` = U holds.[21] This requirement ensures that control transfers to the VMM whenever one of its virtual machines executes an instruction that should be simulated in software. But on some commercially available processors, executing a system-mode instruction while `mode` = U holds does not cause a trap.[22] Instead, the program counter advances, perhaps after instruction-specific changes to memory or registers are performed. These *non-virtualizable* instructions must be handled if we want to implement a virtual machine resembling that real hardware.

Non-virtualizable instructions cease to be problematic if, in any code that a virtual machine executes, we first have replaced each non-virtualizable instruction with code that invokes the VMM. That program rewriting requires:

(i) A means to identify each non-virtualizable instruction and replace that instruction with other code.

(ii) A mechanism to invoke the VMM from within that replacement code.

Various schemes have been devised for (i) and (ii).

Two realizations of (i) are prevalent in practice: binary translation (described next) and paravirtualization (described in §10.4.3). For (ii), many processors include a special *hypervisor call* instruction that, when executed, causes a trap associated with a distinct interrupt class; the corresponding interrupt handler is configured to invoke the VMM. Absent a hypervisor call instruction, the supervisor call instruction (`svc`) discussed earlier would work, provided the VMM can distinguish `svc` executions intended to invoke operating system services from `svc` executions intended to invoke a VMM instruction simulation.[23]

Binary Translation. The process by which binary representation B (the *input executable*) of a program in some *input* machine language is converted into binary representation B' (the *output executable*) for an equivalent program in some *output* machine language is known as *binary translation.*[24] We might want to migrate software that runs on one machine onto new hardware, or we might want existing hardware to execute programs written for hardware that does not yet exist. By taking a liberal view of what constitutes equivalent programs, binary translation can be used to add instrumentation to machine language programs so that run-time behavior will be recorded or measured.

In *static binary translation*, a translator produces output executable B' from input executable B before execution of B' starts. Static binary translation is

[21]System-Mode Instruction Restriction (page 285) introduces this restriction.

[22]The Intel X86 instruction set architecture is a noteworthy example.

[23]A distinguished operand value, never used by the operating system to specify a service, would suffice to distinguish a `svc` execution intended to invoke the VMM.

[24]A machine language program is commonly called a *binary.* So a program to convert between machine languages is doing translation from one machine's binary to another's, hence the name "binary translation".

hard to implement if run-time information determines where instructions start, since the translator would be unable able to ascertain what fragments of an input executable should be converted (because they are instructions) and what fragments should be left alone (because they are data). Uncertainty about instruction boundaries arises when instruction formats are variable-length, instruction alignment has few restrictions, computed branch destinations are supported, and/or instructions are mixed with data.

Dynamic binary translation circumvents uncertainty about instruction locations by converting a block of instructions in the input executable only when that block is reached during execution. By alternating between execution and translation, the translator can read the processor state produced by execution of the last block before converting the next. This processor state not only provides the translator with the starting location for the next block to convert but also can provide values needed for calculating the destination of a computed branch.

Translation and Execution in Alternation. Given is an input executable B, an offset d indicating the location in B for the next instruction to execute, and values to load into processor registers when execution commences.

(1) Construct B' by translating instructions in B, starting at offset d and continuing until a branch instruction ι is encountered whose destination is being computed during execution of B.

(2) Replace branch instruction ι with an instruction that transfers control to the translator. For offset value d passed to the translator, use the offset for ι in B; the processor register values passed are whatever values those registers contain when the translation of ι is reached.

(3) Execute B'. □

Thus, when execution of B' in step (3) reaches the translation of ι, control transfers to the translator (thereby returning to step (1)), which resumes converting B, starting with instruction ι.

VMM use of Dynamic Binary Translation. Dynamic binary translation enables virtual machines to be implemented on a processor whose machine language contains non-virtualizable instructions.

Implementing Virtual Machines by using Binary Translation.

– Implement a dynamic binary translator that replaces system-mode instructions with hypervisor calls. (Recall, non-virtualizable instructions are system-mode instructions, so all non-virtualizable instructions will be replaced by the translator.)

– Implement an interrupt handler for hypervisor calls. This handler should contain code for simulating each system-mode instruction. (That simulation code is already present in VMM privilege interrupt handler $Hndlr_{priv}$ of Figure 10.6.)

– Modify *Dispatcher* for the VMM (page 285) so that its final step transfers control to the dynamic binary translator, providing as arguments the values in the registers of the virtual machine. (The value in the program counter serves as offset d for Translation and Execution in Alternation, above.) □

Dynamic binary translation increases the size of the trusted computing base (by adding the binary translator) and increases run-time overhead (since performing the translation takes time and likely involves making a context switch). A larger trusted computing base seems unavoidable. But we can reduce the run-time overhead by (i) limiting how much of the code is translated during execution, and (ii) not translating the same block of instructions anew every time that block is executed. We now turn to implementing these optimizations.

Dynamic binary translation is not necessary when the following holds.

Binary Translation Elimination Condition. When a non-virtualizable instruction is executed in user-mode its effect is to advance the program counter but not to cause other changes to memory or registers. □

This condition holds for many commecially-available processors. Moreover, the preponderance of code running on a computer will be user-mode; only operating system code executes in system-mode. Thus, when a VMM is implemented using dynamic binary translation on a processor where Binary Translation Elimination Condition holds, then only the operating system code in a virtual machine must incur the run-time overhead of dynamic binary translation.

We demonstrate that when Binary Translation Elimination Condition holds, then executing the input executable is equivalent to executing the output executable for a virtual machine executing in user mode. So producing the output executable is unnecessary. The interesting case is system-mode instructions, given that Implementing Virtual Machines by using Binary Translation does not replace user-mode instructions. There are two cases.

Case 1: A system-mode instruction ι that is non-virtualizable. According to Binary Translation Elimination Condition, execution of ι only advances the program counter when executed on a processor in user-mode. That behavior is equivalent to what would be observed if ι were replaced by a hypervisor call and the hypervisor call interrupt handler simulated the user-mode execution of ι. Dynamic binary translation does exactly that replacement, so execution of ι in the input executable exhibits equivalent behavior to execution of the output executable.

Case 2: Other system-mode instructions. Such an instruction ι will cause a privilege trap when executed, because virtual machines are executed by an underlying processor in user-mode. So, when ι is executed, $Hndlr_{priv}$ of Figure 10.6 receives control and executes a routine to simulate ι. This behavior is equivalent to what would be observed if ι were replaced by a hypervisor call, because the hypervisor call interrupt handler in Implementing Virtual Machines by using Binary Translation (above) simulates execution of ι using code copied from $Hndlr_{priv}$.

A second means for reducing run-time overhead from binary translation is to employ a VMM-maintained *translation cache*; it stores output executables for previously executed (and, therefore, translated) blocks of instructions as well as the values of any registers that affected the translation.

> **Use of a Translation Cache.** For a block of instructions that starts at offset d, a binary translator need not produce an output executable for execution, provided
>
> (i) the required output executable O was previously produced and is available from the translation cache, and
>
> (ii) output executable O in the translation cache is what the binary translator would produce if invoked now. □

Provided (i) and (ii) are cheap to check, Use of a Translation Cache lowers overhead—repeatedly repeatedly executing a block that resides in the translation cache no longer require repeatedly translating that block. When Binary Translation Elimination Condition holds too, the translation cache would store only those parts of the virtual machine's operating system that execute in system-mode; the full performance benefit of a translation cache thus is acheived by incurring only modest storage costs.

To check condition (ii) in Use of a Translation Cache, we can leverage address translation hardware to intercept writes that could create cause cache entries to become outdated.

> **Translation Cache Invalidation.**
>
> – *When an output executable O is inserted into the translation cache:* Disable writes for a region of memory that includes all fragments of the input executable that the translator read when producing O.
>
> – *When a write is attempted to a region of memory where writes have been disabled by the translation cache:* Delete the corresponding output executable from the translation cache; then allow the write to proceed (by invoking binary translation). □

Condition (ii) is satisfied if the output executable is in the translation cache and if current register values equal cached values for registers that affected the translation.

A performance problem with this scheme can arise, however, because address translation hardware typically works at the granularity of memory pages. Far less than a page is read in producing an output executable for a single block of instructions. So writing to a page could cause many output executables to be deleted from the translation cache. Some of those deletions would be unwarranted if only a small part of the page is being updated or if state (but not instructions) is what changed. Such unwarranted deletions can be avoided if the implementation of condition (ii) saves in each cache entry the translator's input and uses that value for later comparison with the contents of memory. This checking of the binary translator's input can be incorporated into the output binary.

10.4.3 Paravirtualization

Transfers of control between a virtual machine and the VMM slow execution by disrupting instruction pipelining and by requiring memory caches to be purged. So performance suffers when a VMM implements system-mode instructions by emulating them in software. In addition, the transparency that makes VMMs so attractive leads to performance problems.

- The operating system in a virtual machine duplicates work performed by the VMM. For example, input/output from an application running in a virtual machine involves executing a driver in the VMM as well as executing a driver in the operating system.
- Work done in the VMM can negate work done in the operating system. Re-ordering of transfer requests that a VMM's disk driver does to enhance disk performance is likely to undermine request re-ordering done by the operating system's driver to enhance disk performance.

Such performance problems suggest that we favor virtual machines where the instruction set does not require software-emulation by the VMM very often.

Virtual machines implemented using *paravirtualization* support (i) the same user-mode instructions as the underlying processor, (ii) a subset of its system-mode instructions, and (iii) a hypervisor call. The set of supported system-mode instructions typically excludes system-mode instructions that are expensive to emulate in software and also excludes all non-virtualizable instructions.[25] VMM-provided hypervisor calls replace the system-mode instructions that no longer are available.

Software comprising user-mode instructions does not have to be changed to run in a virtual machine implemented by paravirtualization. So paravirtualization is transparent to application software. But operating system routines invoke system-mode instructions; that code would have to be changed for execution under paravirtualization. In practice, those changes are typically localized to a handful of routines.

Leverage from Hypervisor Calls. Paravirtualization offers the flexibility to define virtual machines in which hypervisor calls do not simply replicate system-mode instructions. Abstractions well suited to virtualization now can be offered by a VMM. For instance, a clean abstract input/output device is easier to emulate in software than a real device is. So paravirtualization allows a VMM to offer clean abstract input/output devices, resulting in a VMM that is smaller and simpler than one that incorporates emulations for real input/ouptut devices; operating system drivers in virtual machines now can be simpler, too. An abstract input/output device's interface also can be designed to discourage putting functionality in operating system drivers that is duplicated or negated by a VMM's software emulation of the device.

[25]Recall, non-virtualizable instructions are, by definition, system mode.

In addition, if virtual machines employ hypervisor calls to interact with VMM-implemented resources then functionality can be relocated from a VMM into separate, designated virtual machines.

> **Privileged Virtual Machines.** A designated virtual machine V can implement some service for the VMM (and thus for other virtual machines) provided the VMM offers the following.
>
> – The VMM identifies a specific subset of its hypervisor calls as providing a *control interface* for the service.
>
> – The VMM identifies virtual machine V as being *privileged* for the service. V might be, for example, (i) the first virtual machine that the VMM boots or (ii) a virtual machine that boots some specific operating system.
>
> – The VMM ensures that hypervisor calls in the control interface for a service can be invoked only by a virtual machine that is privileged for that service. □

Virtual machines would still use ordinary hypervisor calls for requesting services from the VMM or for retrieving corresponding responses. But instead of the VMM incorporating all of the code to perform that service, a priviliged virtual machine would be involved in processing requests for service; hypervisor calls in the corresponding control interface allow that virtual machine to communicate with its clients. Notice that ordinary virtual machines cannot interfere, because ordinary virtual machines cannot invoke hypervisor calls from a control interface and, therefore, they cannot receive or reply to service requests from clients.

This architecture might seem complicated, and it also expands the trusted computing base to include the operating system and other code that runs in a privileged virtual machine. But it has attractions. First, by relocating functionality from the VMM into virtual machines, we have a basis for increased assurance in the VMM. Second, code that runs in a virtual machine under the auspices of an operating system (with all of its functionality) can be simpler than code that will be incorporated into the VMM. Finally, the architecture allows an existing operating system with existing I/O drivers to provide virtual machines with access to underlying input/output devices. We run this existing operating system in a privileged virtual machine, and doing so avoids the need to write or rewrite input/output drivers for execution in the VMM. Software emulation to create virtualized versions of input/output devices also is now straightforward—virtualized devices can be implemented as servers, benefiting from existing input/output drivers and other functionality that an operating system offers.