Virtualization in x86

Douglas Beal Ashish Kumar Gupta

CSE 548
Winter 2002
University of Washington, Seattle

1 Introduction

Computers are continually increasing in speed, yet they crash all too often. Compelling applications run under different operating systems. Operating system and device driver development is difficult, because errors easily lock the machine, and low-level debugging tools are expensive. Virtual machines solve these problems by providing virtual processors for each operating system to run on. Each virtual processor is isolated, preventing crashes from halting the machine.

The IBM S/390 is the latest machine in a line of mainframes that has architectural support for virtualization. In fact, the S/390's main selling point is its ability to support many simultaneous virtual machines. Consumer level processors lack virtualization support. This complicates any attempt to support virtual machines on top of such processors, possibly resulting in inefficiencies. We posit that architectural support for virtualization will increase speed. The architectural changes that we suggest also support recursive virtual machines, which allows for a layered approach to reach the same functionality.

To examine the speedups possible from architectural changes, we add a Virtual 80386 mode (V386), like the Virtual 8086 mode (V86), to the x86 architecture. We make these changes in Bochs [1] (an open source emulator for x86), and add support into the Linux kernel for a new type of process to take advantage of the V386 mode. We propose a recursive virtual memory scheme that makes recursive virtual machines possible.

1.1 Rest of the Paper

The rest of the paper is organized as follows: Section 2 reviews the fundamentals about Virtual Machines. Section 3 talks about why it is not easy to virtualize current day commercial processors. Section 4 talks about the various techniques that have been used so far to virtualize the processors. In Section 5, we propose architectural changes to x86 that make it more virtualizable and discuss how we can implement these changes. In Section 6, we talk about our future plan of action and finally we conclude in Section 7.

2 Review of Virtual Machines

A virtual machine monitor (VMM) is software for a computer system that creates efficient, isolated programming environments that are duplicates which provide users with the appearance of direct access to the real machine environment. The duplicates are called virtual machines. A VMM manages the real resources of the computer system, exporting them to the virtual machines. The operating system on top of which the VMM is running is called the *host OS* and the operating systems running within the virtual machines are called the *guest OS*s.

2.1 Why Virtual Machines ?

Virtual Machines offer a number of benefits not found in conventional multiprogramming systems. The versatility and flexibility of virtual machines has been successfully utilized in a number

of application areas on small, medium and large-scale computer systems. We will briefly discuss a few of these.

Users and developers can concurrently run different operating systems on the same hardware. Users can run any operating system and applications designed to run on the real processor architecture. Application development becomes easier because a developer can easily test applications on many operating systems simultaneously while running the same base platform.

Operating system debugging becomes a lot easier. Debugging can proceed while the system is being used for normal work. If a VM crashes, the other VMs still keep running.

Each virtual processor is completely secured from interference from any other virtual processor. So, if an untrusted application affects a virtual machine, the integrity of other virtual machines and their applications and data are maintained.

2.2 Some Features of Virtual Machines

A virtual machine is an *efficient, isolated duplicate* of the real machine. The VMM provides an execution environment that is almost identical to the original machine. The behavior of any program executed on a virtual machine should be the same as the behavior when executed on an non-virtualized machine. However, the definition allows for the differences caused by the availability of system resources and differences caused by timing dependencies. This occurs because of the intervening layer of software (VMM) and possibly because of the effect of any other virtual machines concurrently running on top of the same VMM.

The second characteristic of a virtual machine is *efficiency*. A large fraction of the virtual processor's instructions must be executed on the machine's real processor, without any software intervention of VMM. Instructions which cannot be executed directly by the real processor are interpreted by the VMM. This essentially implies that traditional emulators do not fall under the category of virtual machines.

And finally, a VMM must be in control of the system resources. No program running under it in the created environment should have access to any resource not explicitly allocated to it. Also, it should be possible for the VMM to regain control of resources already allocated.

2.3 Components of a VMM

A VMM normally has modules which fall into three groups [10, 9]. The first one is *dispatcher*. A jump to the dispatcher is placed in every location to which the machine traps. The dispatcher then decides which of its modules to call when a trap occurs. The dispatcher can be considered the top level control module of the VMM. It decides which module to call.

The second type of module is the *allocator*. Its task is to decide what system resources are to be provided. In case of a single VM, the allocator needs only to keep the VM and the VMM separate. In case of a VMM which hosts several VMs, it is also the allocator's task to avoid giving the same resource to more than one VM concurrently. The allocator will be invoked by the dispatcher whenever an attempted execution of a privileged instruction in a VM occurs which would have the effect of changing the machine resources associated with that VM. The allocator accounts for most of the complexity of VMM.

The third module type is called the *interpreter*. It can be thought of as an interpreter for the instructions that trap, with one such routine per privileged instruction. The purpose of each such routine is to simulate the effect of the instruction which trapped. This prevents VMs from seeing the actual state of the real hardware. Instead they see only their virtual machine state.

2.4 Types of Virtual Machines

An operating system consists of instructions to be executed on the hardware processor. When an operating system is virtualized, some portion, ranging from none to all, of instructions may be executed by underlying software. Based on the amount of the hardware and software execution

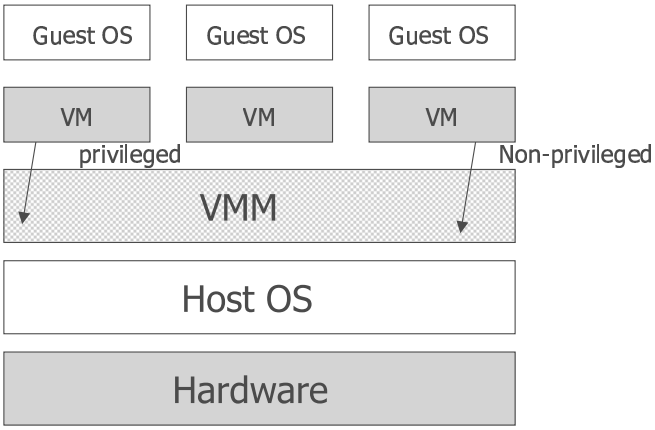


Figure 1: Emulation

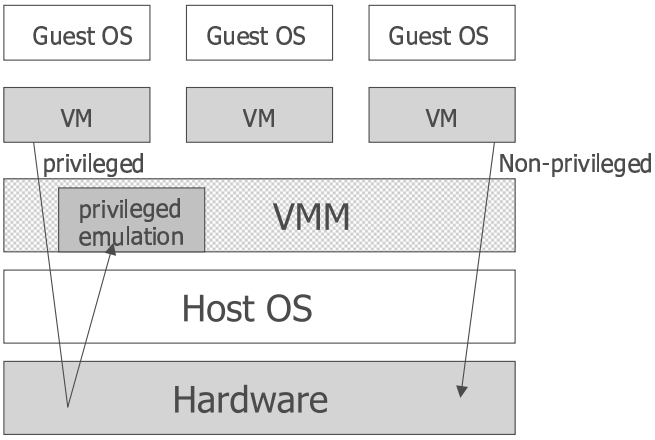


Figure 2: Software Virtualization

of the instructions, we can classify the virtual machines into three categories.

- **Emulation:** The processor is implemented entirely in software. It only uses software interpretation. A software program emulates every instruction of the program (Figure 1). This method has high overhead. Execution of every single instruction in the VM translates into execution of many instructions on the actual processor (as a part of emulation).

- **Software Virtualization:** Only the privileged instructions are emulated. Non-privileged instructions are passed directly to the hardware (Figure 2). Even the privileged instructions are passed directly to the

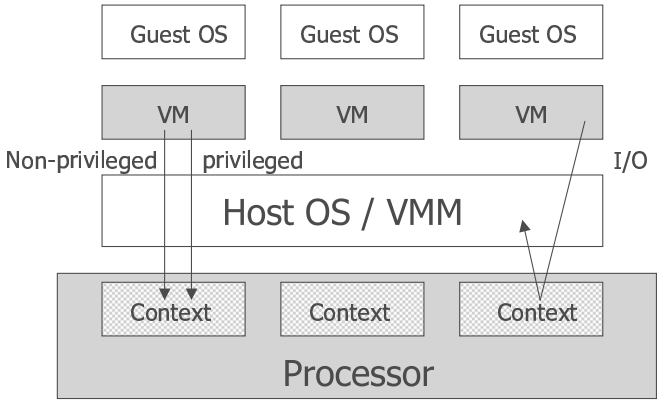


Figure 3: Hardware Virtualization

hardware. However, since the virtual machine runs in the user mode, this generates a trap and the trap is handled by the (*dispatcher* module of) VMM, which then emulates the privileged instructions. Since the I/O instructions are privileged, they are also handled in a similar manner, i.e., the I/O devices are emulated in the software virtualization layer.

- **Hardware Virtualization:** The privileged instructions are no longer emulated in the VMM. They are also executed on the bare hardware (Figure 3). The I/O, however is still emulated in the VMM. There are two ways in which this can be accomplished. In general, we might have a VMM which has a combination of both of these.

  - *Swapping* : If there are not sufficient structures in the hardware to keep track of the context of each concurrent VM, then each time the VMM switches from one VM to another, the context of the former is saved in memory, and the context for the latter is read off from memory. The host OS has the VMM built in it, and the context switching for processes is generalized to handle the VM also. When there is a context switch from one process to the other, if the two processes belong to the same VM, then the usual process level context switch

takes place. However, if they belong to two different VMs, then switching of the VMs is also a part of the context switch between the two processes.

- *Duplicated structures* : The processor has enough hardware resources to keep track of the context of all the VMs that are running.

3 What complicates virtualization?

In order to provide the applications running on top of the VM with the appearance of direct access to the real machine environment, a VMM needs to virtualize all the resources that would have been available to such an application, if it were running in a real machine environment. These resources can be divided into two categories (a) Processor Resources (b) I/O Devices. Below, we briefly discuss each of these.

3.1 Processor Resources

To virtualize a machine, the first thing that needs to be virtualized is the processor. An important point to note here is that there is no need to virtualize all of the processor resources. It is the application that decides which processor resources have be to virtualized. Choosing a subset of the processor functionality to virtualize can simplify the logic of the VMM. For example, Denali provides para-virtualization, where a subset of the x86 ISA is provided and no virtual memory is provided to guest OSes [11]. However, to be able to run all operating systems without any modification, all the processor resources need to be virtualized.

Most processor architectures contain two or more privilege levels - a feature that can be exploited when virtualizing the processor. Typically, the most privileged level is used by the operating system and driver software. Application software, on the other hand, uses the least privileged level (sometimes called the *user mode*). These levels limit access to certain functionality within the processor, allowing the operating system to maintain control over the system at all times.

Some of the instructions executing within a VM cannot be directly executed on the processor. These instructions are called sensitive instructions. The key to implementing a VMM is to prevent the direct execution of sensitive instructions.

If all the sensitive instructions of a processor are privileged, then the processor is considered to be virtualizable. When the sensitive instructions are executed in user mode, all sensitive instructions will trap to the VMM. After trapping, the VMM will execute code to emulate the proper behavior of the privileged instruction for the virtual machine, while still maintaining its control over the system. However, this technique relies on the fact that most processors faithfully generate exceptions for all sensitive instructions when executed in user mode. But, if sensitive non-privileged instructions exist, it may be necessary for the VMM to examine all instructions before execution to force a trap to the VMM when a sensitive, non-privileged instruction is encountered. This incurs a very severe performance penalty.

The x86 architecture violates the *all sensitive instructions are privileged* rule. For example, the x86 makes use of a "global descriptor table" (GDT) to define segments through which all memory accesses are performed. Because the GDT is a global resource, it must be maintained by the operating system. It would logically follow that all processor instructions that modify or explicitly access the GDT should be considered privileged, and therefore inaccessible from user-mode code. The x86 architecture does treat the LGDT (load global descriptor table) instruction as privileged, but the SGDT (store global descriptor table) instruction is not. Because the SGDT instruction doesn't cause an exception in user mode, there is no opportunity to virtualize the location of the GDT through the use of exception handlers. If a program running on top of a VM were allowed to look at the value of GDT, it would end up reading the wrong values. The

state stored in the GDT corresponds to that of the VMM running on top of the processor and not of that program. A study of the x86 instruction set conducted in [10] concludes that there are 17 such instructions.

3.2 I/O Devices

Most I/O devices are built to be driven by a component of an OS called the driver. No other software must interfere with this interface, otherwise there will be conflict. As a result, the virtualized guest OS cannot drive the device that is already being driven by the Host OS. Thus, the VMM has to intercept all the I/O accesses from the CPU and emulate a complete set of devices in software, such that the I/O instructions believe they are getting such information from the real devices. Many alternatives are available to go about virtualizing hardware devices. The choices affect performance, and ease of implementation. Here we look at four such possible ways.

- **Static Partitioning:** Some VMMs have side stepped the issue by statically allocating hardware to VMs [6]. This method can provide almost native performance.

- **Lowest Common Denominator Emulation:** Some VMMs emulate a lowest common denominator device. For example, Plex86 [8] emulates a NE2000 network card. The advantage of emulating an existing device is that the guest OSes can run without any modification, and all OSes have drivers for the device. This is because the interface presented by the emulated device will mimic the interface presented by the real device. So, the device drivers in the OS that work with the real devices will work with these emulated devices too. However, this can be a source of inefficiencies because of the constraints and the behavior imposed by the real hardware device.

- **Virtual Device Emulation:** The VMM does not replicate the behavior of the real hardware devices, instead it provides a virtual device at a higher abstraction level. The VMM is no longer constrained by the behavior of the real devices. It dictates the semantics of the device, making it more efficient. The translation of virtual device interactions into interactions with real devices can either take place within the VMM itself or within the host OS. In Denali [11], the VMM does this translation, where as in VMWare [4], the translation is done in the Host OS. The drawback of this virtual device emulation scheme is that a driver for the virtual device needs to be written for each host operating system.

- **I/O API:** At the highest level of abstraction, a VMM can provide an I/O API. Fluke [7] takes this approach. In Fluke, the I/O API allows arbitrary layering of VMs to provide services.

4 Techniques used for Virtualization so far

There has been a great deal of work in the area of virtual machines in the past. The idea of a virtual machine actually goes back almost to the beginning of computing itself. The work done in the area of virtual machines can be broadly classified into two categories. One, where the support for virtualization is provided in the software layer. The other is when the virtualization support is provided in the hardware.

4.1 Software Virtualization

The first technique we discuss here is *Scan Before Execute (SBE)* [8]. The basic idea is not to let the execution of the code from guest OS pass through unscanned code. This is to avoid the execution of any sensitive non-privileged instructions directly on the bare hardware. These instructions are emulated in software and the others are executed on the hardware. *Dynamic Scan Before Execute* [8] is a more efficient version of this technique. Finally, we look at *Para Virtualization* [11].

- **Scan Before Execute:** A scan phase precedes the execution of the code. The entire code page is scanned first and then a virtualized code page is created that mirrors the real code page. In the virtualized code page, a breakpoint is placed before sensitive non-privileged instructions so that they can be emulated in the software. When the code is executed from the virtualized code pages, a breakpoint exception is raised when the inserted breakpoint is encountered. The VMM handles the exception by emulating the instruction in software, and then control returns to the point after the software breakpoint.

- **Dynamic Scan Before Execute:** This is a more efficient version of SBE. SBE suffers from the drawback that it scans every page of code irrespective of whether that code is ever executed or not. Dynamic SBE overcomes this. It is same as SBE, but works on a page-by-page code basis. It fetches and decodes a sequence of instructions up to a branch instruction and places a breakpoint there. When the code gets executed, a breakpoint exception is generated at the branch instruction. The VMM receives the exception, effects the branch and given the new target address after the branch, repeats the same process for the next code sequence. Similar breakpoint exceptions are placed before the sensitive non-privileged instructions as well. VMWare [4] is a commercially available virtualization software program, that works on similar lines.

- **Para Virtualization:** Since virtualizing every processor resource is difficult, virtualize only that subset of the resources that are required by the application. For example, Denali [11] virtualizes only a subset of the x86 ISA.

4.2 Hardware Virtualization

One of the earliest known virtual machine system is IBM's VM/370 [5]. On each virtual 370, a user may run any of the System/360 or System/370 operating systems. The user may also run a simple multi programming system which was developed specifically for use on virtual machines. A more recent mainframe with support for virtualization is S/390. It has specialized circuits in the CPU to allow it to virtualize itself. The S/390 actually virtualizes itself in hardware, meaning that applications run at full native speed except for a very, very few privileged instructions that are actually emulated by the operating system. Everything is virtualized, right down to the hardware I/O addresses and memory map.

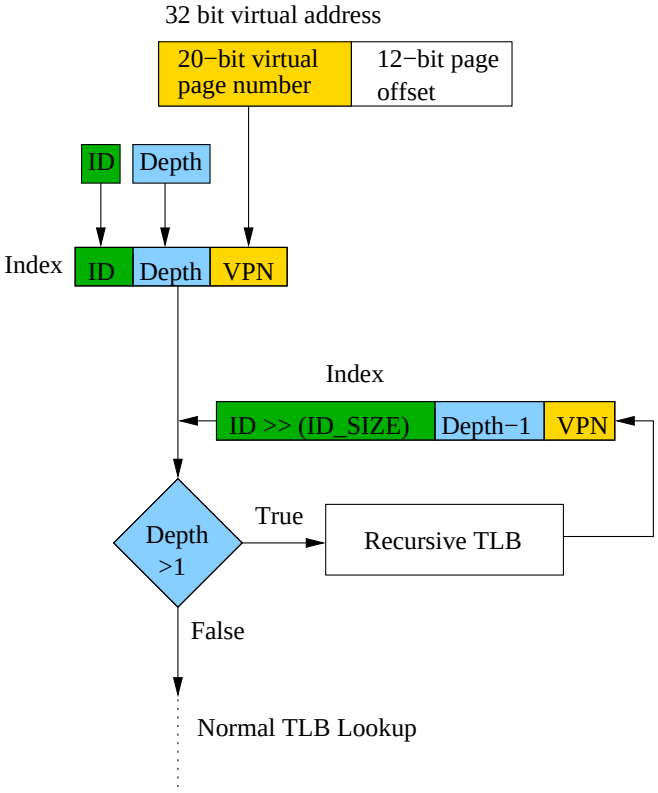
5 What we are doing

The x86's sensitive non-privileged instructions make virtualization difficult and overhead intensive. Even if these instructions were fixed, there is still a significant overhead in emulating privileged instructions. A natural approach is to push all the overhead down into the processor to avoid software virtualization overhead. To add support for virtualization in the hardware, we need to modify the x86 processor. Exercising the changes requires software to take advantage of the new features. These factors lead us to create a virtual processor mode, and modify the Linux Kernel to take advantage of the new mode.

5.1 Virtual 386 Mode

x86 processors newer than the 80286 already have a virtualization mode built in. Virtual 8086 mode allowed newer processors to run a virtualized 8086, even while they were in protected mode (*had virtual memory turned on*). Older versions of Windows used the Virtual 8086 mode to run DOS programs, and DOSEMU [2] uses Virtual 8086 mode under Linux to run DOS programs. We extend the idea of a Virtual 8086 mode into virtual memory by adding Virtual 80386 mode. Unlike the relatively simple Virtual 8086 mode, Virtual 80386 mode must deal with the more complex memory addressing modes of the 386 instruction set, external interrupts and I/O instruction

Figure 5: Recursive Virtual Memory Lookup



emulation. Further more, to allow recursive virtual machine architectures that layer functionality in different virtual machines, such as Fluke [7], we support recursive virtual memory.

5.1.1 Recursive Virtual Memory

The largest processor change to support Virtual 80386 mode is the addition of another TLB for translating virtual machine virtual memory to real virtual memory. To support recursive virtual machines, the additional TLB must allow recursive lookups. Figure 4 shows a recursive lookup for the shaded page belonging to *Virtual Machine 0xFEFC (with designator 0xC)*.

An efficient implementation of the iterative lookup requires an additional TLB that has the virtual machine memory mappings. To differentiate between different addresses generated by different virtual machines, a *Virtual Machine Identifier* is included. A virtual machine level is also included to determine when the recursive lookup

Figure 6: Recursive TLB - One Lookup Path

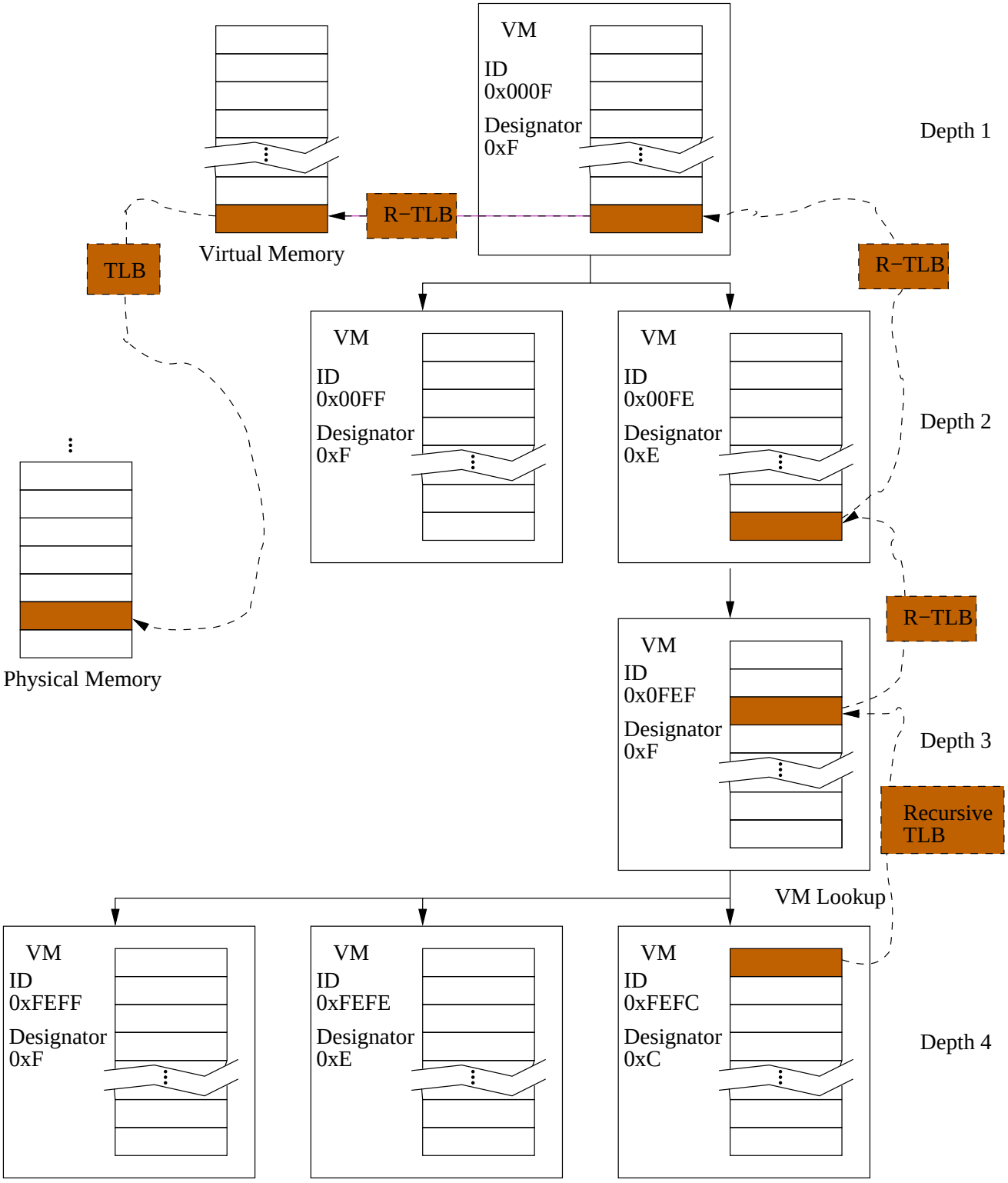
ID	Level	Input Address	Output Address
FEFC	4	FEFE	ABCD
0FEF	3	ABCD	1234
00FE	2	1234	6789
000F	1	6789	EEEE

has reached a real virtual address to feed to the standard virtual memory lookup. The identifier consists of a virtual machine designator (a local ID) chosen by the launcher and the designator's of all parent virtual machines. A virtual machine will not be able to determine that it is virtual unless it performs an attack to exhaust the ID space, as the hardware is responsible for appending the parent designators to generate the full virtual machine ID. Figure 4 illustrates a possible virtual machine hierarchy, with the parent-child relationship indicating that the parent launched the child. The root node was launched by the base operating system. An important implementation detail is the virtual machine ID partitioning. For example, a 32-bit identifier with 4-bits for virtual machine designator allows each virtual machine 16 children, and allows the maximum hierarchy depth 8.

Figure 5 shows the logical flow of the recursive TLB lookup. The 20 high order bits of the virtual address are combined with the virtual machine ID and depth. If the depth is 1, then a normal TLB lookup is performed. Otherwise, they are used to find an entry in the *Recursive TLB*. If no entry is found, the processor walks the OS page table looking for the entry. The OS page table only contains the mapping for one level of the page table. The depth is decremented, and the ID right shifted by the designator size, and the lookup is performed again. If a page table other than the currently active one needs to be walked, then that context needs to be switched in. When a page actually faults, the fault needs to be serviced by each virtual machine in the hierarchy, so they can allocate the page in their tables.

Figure 6 shows an example lookup path four levels deep, where all the entries are present in

Figure 4: Recursive Virtual Machine



the recursive TLB. Figure 4 also shows a recursive lookup from the virtual machine with ID 0xFEFC. First, the virtual machine generated an address in the shaded page. The processor takes the first 20-bits of the address, the depth (4), and the ID (0xFEFC) and uses them to index the *Recursive TLB*. The depth is decremented (3), and the ID is shifted by the designator size (0x0FEF). Since the depth has not yet reached 1, another lookup is performed. This lookup results in an address in 0x00FE. The lookup process is repeated until the depth reaches 1, which indicates that the *Recursive TLB* value is a real virtual address. The real virtual address is used to index the normal TLB to get the physical address. Now that the processor has determined the physical address, the address is resolved.

Paging virtual machine memory out to disk requires the launching OS to free the paged page. Since the launching OS treats the page as physical, it has no concept of freeing the page, even though the page is allocated by the parent virtual machine. This means that pages may be paged out at the root level, but they won't be freed. To counteract this effect, the launching OS must inform child virtual machines of a limit on available memory (most likely through BIOS emulation). For system stability, this should be a hard limit.

5.1.2 Interrupts

Interrupts also need special handling. The host OS needs to be able to preempt the virtual machines in Virtual 80386 mode. This means that there must be interrupts/timers that are inaccessible to the virtual machines (note that they must appear to work for the virtual machines, otherwise they will be able to detect that they are virtual machines) to effect a context switch back to the host OS. The virtual machines need memory mapped I/O to interact with hardware, which is emulated by the host OS when the I/O instructions trap. External interrupts also need to bring control back to the host OS.

5.1.3 Virtual 80386 Mode Interface

Activating Virtual 80386 mode is accomplished in the same manner as Virtual 8086 mode, by turning an EFLAGS bit on. Additionally, the Virtual 80386 subsystem needs to know the virtual machine designator for recursive virtual memory lookups. The designator is communicated to the processor by placement on the stack, which avoids changing any virtual machine state.

5.2 Virtual 80386 Implementation Platform

To prototype Virtual 80386 mode, we are modifying Bochs [1]. Bochs provides a modifiable x86 ISA as an Open Source x86 emulator. To take advantage of the Virtual 80386 mode, we are modifying a Linux kernel so it can launch new processes in Virtual 80386 mode inside of Bochs. While Bochs provides a platform to easily prototype x86 ISA changes, further work is needed to determine how costly these changes would be in hardware.

5.2.1 Virtual 80386 Support in Linux

To test the Virtual 80386 mode in Bochs, software capable of launching new virtual machines, dealing with memory allocation, and emulating I/O is necessary. We chose the Linux kernel as the starting point, as it already has similar code to take advantage of Virtual 8086 mode. We added a system call that takes a drive image file and a bootstrapping program, and launches the bootstrapping program with an emulated drive using the image file. The Virtual 80386 mode uses loadable kernel modules to provide device emulation. Existing process switching code is used to save processor state when switching out of Virtual 80386 mode, with a small modification to enable the Virtual 80386 EFLAGS bit.

6 Future Work

Once we finish the prototype, we want to compare it against VMWare. Running both virtual-

izer platforms with benchmarks [3] under Bochs allows us to capture an instruction trace. We can then analyze the instruction trace and compare relative performance. We expect that Virtual 80386 mode will outperform VMWare for I/O because our system emulates I/O devices in the kernel, obviating the need for a context switch. However, VMWare's method of hacking up the host OS's page table may give it an advantage when allocating new pages. Access to page tables should be similar, because in VMWare they are all unified, and in Virtual 80386 there is a Recursive TLB. Virtual 80386 performance will degrade significantly in low memory conditions with recursive virtual machines, but VMWare doesn't support recursion.

While implementing a Virtual 80386 mode using Transmeta's code morphing technology is very attractive, they don't intend to allow general development with it. A path more likely to result in an actual processor is modifying a different architecture to have a virtual mode. Unfortunately, simulators such as SimpleScalar gloss over operating system details by emulating system calls directly. Unfortunately, this makes them unsuitable as platforms to implement a virtual mode on.

7 Conclusions

Virtual 80386 mode is attractive for several reasons. It makes it easy and safe to run multiple different operating systems side by side. It makes the development of new operating systems easy, because you have a stable base to launch and debug the experimental operating system from. It allows untrusted code to be run with little trust from the user (you have to trust the I/O emulation code). The Virtual 80386 mode also makes the development of VMMs easier, and can be used for the interesting VMM ideas like code migration and check-pointing. Fortunately, all of these ideas can be run, at a performance penalty, on the Bochs Virtual 80386 mode implementation without a hardware realization.

References

[1] Bochs IA32 emulator. http://bochs.sourceforge.net.

[2] DOSEMU linux x86 DOS emulator. http://www.dosemu.org.

[3] Linux benchmark suite. http://lbs.sourceforge.net.

[4] Vmware, inc. vmware virtual machine technology. http://www.vmware.org.

[5] IBM virtual machine facility/370 planning guide. *IBM Corporation Publication No. GC20-1801-0*, 1972.

[6] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1), 1989.

[7] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[8] K. P. Lawton. Running multiple operating systems concurrently on an ia32 pc using virtualization techniques. http://www.plex86.org/research/paper.txt.

[9] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, August 1974.

[10] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. *Proceedings of the 9th USENIX Security Synposium*, August 2000.

[11] A. Whitaker, M. Shaw, and S. D.Gribble. Denali:lightweight virtual machines for distributed and networked applications. Submitted for publication, 2002.