Improving the Latency of VMExit Forwarding in Recursive Virtualization for the x86 Architecture

Wing-Chi Poon  Aloysius K. Mok
VMware, Inc. University of Texas at Austin
wpoon@vmware.com mok@cs.utexas.edu

Abstract
As virtualization becomes more common, the need for recursive virtualization is also rising. However, performance degrades exponentially as we go deeper in the number of levels in recursive virtualization.

In this paper, one of the biggest challenges of improving the performance of recursive virtualization is explored, namely bounding the time for VMExit forwarding across the hierarchy of hypervisors, including VMExits due to interrupts, exceptions and various other intercepts. The problem is analyzed in the context of the current x86 architecture. We argued that a purely software solution will likely fall short, and then proposed a simple hardware extension that could speed up the VMExit delivery significantly.

1 Introduction

1.1 Definition
A Virtual Machine (VM) is an abstraction of an execution environment where system and/or application programs can be run. The piece of software that provides this interface is called a Virtual Machine Monitor (VM Monitor) or hypervisor. There are many flavors of VM Monitors: some present a different machine interface from that of the host (e.g. the Java Virtual Machine (JVM) [14]), others provide an identical machine interface (e.g. VMware Workstation or ESX server).

Just as an OS abstracts the underlying hardware details to provide a software interface for simultaneously running one or more application programs, a hypervisor virtualizes the underlying hardware interface to provide (virtual) instances of the hardware interface to simultaneously run one or more OSes, commonly referred to as guests. Inside a hypervisor, each guest OS is run in a separate VM instance. A VMEntry is an event of execution transition from the hypervisor to the guest, whereas a VMExit is an event of execution transition from the guest back to the hypervisor. VMEntry and VMExit are collectively called world switches.

In theory, a VM instance should be able to run another copy of the hypervisor, resulting in recursive virtualization (also called nested virtualization). A hypervisor participating in recursive virtualization is called a hierarchical hypervisor. In theory, there could be an arbitrary number of hierarchical hypervisors participating in recursive virtualization between the actual hardware and the ultimate guest OS.

The hypervisor that is the closest to the hardware is called the bottommost level (level 1), and the one that is the closest to the OS the topmost level (level n). For different OSes in the same system, the topmost level may reside at a numerically different level number. From the point of view of level i, level i – 1 is the parent level, level i + 1 is the next level or child level, all successive parents levels are collectively called ancestor levels, and all successive children levels the descendant levels.

1.2 Related Works
Machine virtualization has been studied for a long time [10]. Popek and Goldberg [16] listed separate criteria for a machine architecture to be virtualizable and recursively virtualizable at the system level. Specifically,
the AMD/Intel x86 CPU architecture has been shown to be not virtualizable [17] according to the requirements. Despite that, a number of techniques have been used in various successful attempts to virtualize the x86 architecture, notably by VMware and Xen. VMware also has begun to support running VMware ESX server or Workstation inside another VMware ESX server or Workstation.

The problem of recursive virtualization has been tackled in various ways before. Some adapted the machine-OS interface [13] [5] [11] while others tackled the OS-application interface [8] [2] to make way for recursive virtualization. This paper bases the hypervisor on unmodified x86 hardware, taking an unmodified OS (e.g. Linux, Microsoft Windows) or “bare-metal” hypervisor as guest.

Before the arrival of the hardware-assisted virtualization technology, virtualization of the x86 architecture was done with binary translation [1]. From the point of view of a hypervisor, a guest binary translator is a piece of self-modifying code, which is known to interact poorly with the binary translator in the parent hypervisor, because binary translating a self-modifying code means frequently throwing away translated code cache and re-translating whatever were just self-modified.

Both AMD [25] and Intel [22] introduced hardware support for virtualization. Although the present hardware only supports a single level of virtualization, they are extensible to recursive virtualization. The advent of hardware-assisted virtualization technology enabled performant implementation of recursive virtualization because most of the guest kernel code could also be run directly in hardware.

Adams & Agesen published a performance comparison of binary translation vs. hardware-assisted virtualization [1]. From their analysis, it is clear that the performance bottleneck of hardware-assisted virtualization is dominated by the latency of VMEntry and VMExit. Newer hardwares addressed this latency issue, as well as introduced nested paging to vastly reduce the frequency of VMExits.

However, the effect of a long latency in VMExit is greatly magnified when it comes to recursive virtualization. Hardware always deliver VMExits to the bottommost-level hypervisor. It is up to the hierarchy of hypervisors to forward each VMExit to the correct level before it could be properly processed. While a VMExit is being processed, it generates more VMExits that need to be similarly forwarded, resulting in an avalanche of VMExits.

1.3 Our Contribution

In this paper, different types of VMExits are categorized as either forward propagation or reverse propagation in recursive virtualization. A software solution for forwarding VMExits is formulated and its running time analyzed to show that a purely software solution is insufficient. Then a possible hardware extension is proposed to improve the performance. All timing analysis are verified with simulation.

2 Why Recursive Virtualization?

Recursive virtualization is gaining traction nowadays. The IBM Turtles Project [4], VMware, Xen [2] and KVM [23] all support some form of recursive virtualization. In this section, we will discuss the motivation for recursive virtualization.

2.1 Modern OS Incorporates Virtualization

Modern OS starts to incorporate virtualization functionality. For example, Microsoft Windows 7 runs XP mode in a virtual machine. It takes recursive virtualization for a hypervisor to run such newer OS.

2.2 Debugging and Upgrading to New Hypervisor

A hypervisor could be used during the development of an OS to probe into the machine state, capture and restore the machine state, etc. It could be similarly extended to development of hypervisors. A hypervisor to debug another hypervisor means recursive virtualization.

When a hypervisor is upgraded, it is desirable to keep both versions running for an extended period of time while guests are migrated, so as to make sure that the new one performs correctly, or at least as good as the old one. Recursive virtualization is needed if two versions of hypervisors must be up and running simultaneously on a single host for comparison.
2.3 Testing Hypervisor Management Software

A specific example, encountered at VMware, is the need to test hypervisor management software at a large scale. Large numbers of hypervisors must be managed. Without recursive virtualization, this requires an expensive setup with as many hosts as hypervisors. With recursive virtualization, significant hardware cost savings can be achieved by running multiple copies of VMware ESX servers within another VMware ESX server or Workstation.

2.4 Hardware Feature Prototyping

The introduction of new hardware features is currently prototyped in simulators, which are often slow, and do not simulate all hardware features (e.g. CPU simulator may not simulate cache effects and hence ignores instructions like WBINVD that invalidates the cache). If a new hardware feature is prototyped in a hypervisor layer, real systems could be tested faster and the performance results obtained may be more representative of the real system. If a hardware prototyping hypervisor is able to run a modern OS and hypervisor, it itself has to support recursive virtualization. Hardware features that can be prototyped this way include but are not limited to new instructions, more memory or processors, and enhancements to I/O controller units.

3 Design Issues

3.1 Statically Determined Interrupt and Exception Handling Sequence

In recursive virtualization, it is observed that the level which gets to handle an exception or an interrupt is always well defined. Basically, if the exception or interrupt is generated internally by software, it is handled top-down (from higher to lower numerical level number); but if the exception or interrupt is generated externally, it is handled bottom-up (from lower to higher numerical level number).

Specifically in the context of the x86 architecture, all fault-type (including hidden page fault #PF in shadow paging), trap-type, abort-type exceptions (except machine check exception #MC), all instruction intercepts, all I/O interrupts, all software interrupts (INT) and processor shutdown (triple-fault) are internal events; while all non-maskable interrupts (NMI), system management interrupts (SMI), maskable external interrupts (INTR), external processor initialization (INIT), machine check exception #MC and processor freeze (FERR) are external events.

3.2 Forward Propagation

When a hypervisor receives an external event, it either consumes the event itself, or needs to forward the event to the correct guest in the hierarchy. This is called forward propagation. Refer to figure 1, and suppose the processor is currently assigned to App 2 while the keyboard is assigned to App 1. Now, external timer interrupts should be forwarded to OS 2 while at the same time external keyboard interrupts should be delivered to App 1.

3.3 Reverse Propagation

Internal events should be delivered in a top-down manner. However, with current hardware, such events are always

---

1 An SMI could be caused by internally trapping I/O instructions, or asserted externally. Ideally, it should be handled top-down in the first case, and bottom-up in the second case.

Debug exceptions #DB is another special case. When the use of recursive virtualization is to debug a new hypervisor or OS, the bottom-most hypervisor may wish to own the exception together with the debug registers, in which case the exception should be handled bottom-up. In all other cases, the hypervisor should leave the debug registers to its guests, and handle the exception top-down.

---

5606
delivered to the bottommost hypervisor first. Thus the hierarchy as a whole needs to simulate a top-down delivery of such events. In figure 2, exceptions generated in App 1 should be delivered to OS 1, and those from OS 3 should first be triaged by hypervisor 3, as shown by the dotted arrows. However, real exceptions travel according to the solid arrows, with hypervisors jointly forward each exception back to its correct level. This is called reverse propagation.

As each intermediate hypervisor processes an event (both internal and external), it generates more (internal) events that are reverse propagated to the next lower level hypervisor. This pattern continues until the bottommost hypervisor finally gets to process the event.

3.4 Performance Measurement Methodology

The performance of each proposed algorithm is first analyzed mathematically and then simulated empirically. Current hardware is not performant enough for deep nesting, so the mathematical analysis is verified with simulation in this paper. The approximate number of clock cycles spent in interrupt / exception delivery is graphed against the hypervisor level number for each of the proposed algorithms (figures 3 and 4). The mathematical analysis is shown as solid line whereas the empirical measurements from simulation are shown as data points with error bounds.

Both the mathematical analysis and the empirical simulation require some data about the speed of certain hardware instructions (e.g. IRET) and events (e.g. #GP exception). Some of the numbers were obtained from AMD published data [24], and measurements were performed to determine the rest. All measurements were done on a 6-core 2200MHz SVM-enabled AMD Istanbul processor (Opteron 2427). The numbers obtained are recorded in table 1.

<table>
<thead>
<tr>
<th># of clock cycles</th>
<th>AMD [24]</th>
<th>FrobOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU mem reg/imm</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ALU reg reg/imm</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BT mem imm</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>BT mem reg</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>#GP exception</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>INC reg</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IRET</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Jcc/JMP disp</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LIDT</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>MOV mem reg/imm</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV reg reg/imm</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV reg mem</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV reg SS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PUSH mem/addr</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>VMLOAD</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>VMSAVE</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>(round-trip) World Switch ‡</td>
<td>794</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Characterization for AMD Istanbul (family 10h)
† ALU instructions include ADD, AND, CMP, OR, SUB, XOR, etc.
‡ Round-trip world switch time is measured as the combined time for a VMRUN instruction followed immediately by a VMExit event that is triggered by an intercepted #GP exception in the first guest instruction.

A custom OS (FrobOS, that VMware developed for VMM testing) is re-purposed to measure the number of clock cycles these instructions or events take natively in 64-bit long mode. This hardware characterization test is extended from the same nanobenchmark used by Adams & Agesen [1]. Wherever the measurement overlaps with AMD’s (e.g. IRET instruction), the measured results are in agreement with AMD.

In the simulation, the algorithms are rewritten in assembly code and then implemented with each x86 instruction converted to a function that accumulates the number of simulated clock cycles spent according to table 1. Instructions or events that require reverse propagation are converted into recursive function calls that follow the actual propagation sequence while accumulating the simulated clock cycles. Randomization is used when control flow depends on external factors.

5607
The simulation was run 65536 times for each algorithm at nesting levels 1 to 5, and the minimum, average and maximum number of clock cycles spent were plotted, overlaying the mathematical analysis result.

4 VMExit Forwarding under Current Hardware Support

AMD has Secure Virtual Machine (SVM, also known as AMD Pacifica Technology) while Intel has Virtual Machine Extension (VMX, also known as Vanderpool Technology Extension, or VT-x). They provide direct hardware support for a single-level hypervisor.

4.1 Intercept Handling

A hypervisor using current hardware support can specify precisely which events to intercept in the VMCB Control Block or VMX Controls in VMCS. An intercepted event results in VMExit, which is handled by the code immediately after the VMRUN (for AMD) or the instruction specified in the VMCS when invoking VM_LAUNCH/VM_RESUME instruction (for Intel). The guest is not restarted until the hypervisor executes VMRUN or VM_RESUME again.

This architecture could be extended to recursive virtualization in two ways. In the first approach, paravirtualization would call for an omnipotent bottom-level hypervisor that does all the work and keeps track of all state information for all levels. However, the second approach is preferred, where each hypervisor takes care of only its next level (i.e., immediate guests). According to section 3.1, any event is either forward propagated or reverse propagated across the hierarchy (see figure 2). Here the pseudo-code for reverse propagation is presented. reverse_propagation_svm() is for AMD's SVM Architecture. The corresponding one for Intel’s VMX Architecture, and the ones for forward propagation under both architectures are very similar.

\[\text{reverse\_propagation\_svm}()\]
\[\text{initialize \ gVMCB}\]
\[\text{clear proxy flag}\]
\[\text{while (true) \{\}
  \text{RAX := proxy ? pVMCB : gVMCB}\]
\[\text{VMLOAD}\]
\[\text{while (true) \{\}
  \text{restore additional registers}\]
\[\text{VMRUN}\]
\[\text{save additional registers}\]
\[\text{if (handling VMExit is easy) \{\}
  \text{handle VMExit}\]
\[\text{else break}\]
\[\text{\}\}
\[\text{VMSAVE}\]
\[\text{if (proxy) and (guest intercepts this) \{\}
  \text{clear proxy flag}\]
\[\text{gVMCB.State := pVMCB.State}\]
\[\text{\}\]
\[\text{else if (guest trying to VMRUN) \{\}
  \text{set proxy flag}\]
\[\text{gVMCB.rip := gVMCB.rip + 3}\]
\[\text{pVMCB.Ctrl := gVMCB.Ctrl}\]
\[\text{bitwise-or gVMCB.rax->Ctrl}\]
\[\text{pVMCB.State := gVMCB.rax->State}\]
\[\text{\}\]
\[\text{else handle other VMExit}\]
\[\text{\}\}
\]

the guest’s gVMCB/gVMCS (pointed to by gVMCB.rax upon VMExit when the guest tries to do VMRUN, or given in VMX-Instruction Information Field of VMCS upon VMExit due to guest’s attempted execution of VMPTRLD). In pVMCB/pVMCS, anything that the guest or the hypervisor wants to intercept are intercepted. The proxy flag is used to distinguish whether the hypervisor is running the guest or running an image on behalf of the guest.

If the hypervisor is running a guest, it deals with whatever VMExit it catches. But if the hypervisor is running an image on behalf of the guest, it appropriately decides whether to forward the VMExit event to the guest handler or consumes the event itself. If it wants to forward the event to the guest handler, it simply re-starts the guest at the instruction following VMRUN or at the location specified in VMCS when invoking VM_LAUNCH/VM_RESUME, with the gVMCB/gVMCS updated with the state information from pVMCB/pVMCS.

Since it is mandatory to intercept VMRUN in AMD SVM and VM_LAUNCH/VM_RESUME in Intel VMX, the hypervisor would only proxy VMRUN or VM_LAUNCH/
VMRESUME for its immediate guest. Hence it takes care of only the next level in the hierarchy. When a level 3 hypervisor tries to VMRUN or VMLAUNCH/VMRESUME, the level 1 hypervisor intercepts it and forwards it to the level 2 hypervisor. The level 2 hypervisor then sets up a proxy VMRUN or VMLAUNCH/VMRESUME for the level 3 hypervisor, which is again caught by the level 1 hypervisor. And now the level 1 hypervisor sets up a proxy for the level 2 proxy. The level 1 hypervisor has no knowledge that the VMRUN or VMLAUNCH/VMRESUME of the level 2 hypervisor it tries to proxy for is itself a proxy for the level 3 hypervisor!

4.2 Running Time Analysis

We analyse the time it takes to propagate a VMExit. The actual time spent inside the handler to service the VMExit does not affect the effectiveness of the propagation. Thus the round-trip time to a null handler is determined to represent the effectiveness of the propagation algorithm.

For the bottommost hypervisor, the running time is dominated by the VMExit and VMEnter events. These are the events that involve heavy weight world switch between the hypervisor and the guest. Let $t_{WS}$ be the time it takes for each world switch: $T_1 = 2t_{WS}$. For all higher levels, the entry point when an intercept is forwarded to the next level hypervisor is the VMExit event at line 10 (immediately following the VMRUN instruction) in reverse_propagation_svm(), and the running time is measured until control loops back to the VMRUN instruction at line 9. The VMSAVE, VMLOAD and VMRUN instructions on lines 15, 6 and 9 respectively require reverse propagation of their own. Thus, $T_n = 3T_{n-1} = 3^{n-1} \cdot 2t_{WS}$.

From table 1, $t_{WS} = 794$ cycles. The equation is graphed as solid line in figure 3, overlaid with simulation results as data points with uncertainty range.

The mathematical analysis closely matches but slightly underestimates the running time than the simulation result. This is because only the key steps that take up the largest number of clock cycles in the mathematical analysis are considered and the rest is ignored.

If a purely software solution is used with the current hardware-assisted virtualization support, latency for propagating VMExits in recursive virtualization is exponential in terms of the level number that the intercept needs to be forwarded to.

5 Hardware Extension to Support Recursive Virtualization

The whole VMExit forwarding could be done better if a simple hardware algorithm is adopted. This algorithm accomplishes correct delivery of VMExit to the hypervisor at the correct level.

5.1 Ancestor and Descendant Linked Lists

First of all, hardware needs to keep track of the chain of hierarchical hypervisors loaded at any moment. This can be achieved by adding a pointer to the VM_HOSTSAVE_AREA (for AMD) and VMCS Host-State Area (for Intel) to point back to their parent VM_HOSTSAVE_AREA or VMCS Host-State Area respectively. Thus, no matter which level in the hierarchy is currently running, this pointer chain links all the ancestor VM_HOSTSAVE_AREA or VMCS Host-State Area in a linked list. Similarly, from the parent area, hardware can find the next level by keeping their currently running VMCB or VMCS Guest-State Area in their parent’s area.
5.2 Intercept Redirect Bit

Instead of forcing all intercepts to statically fall into either forward propagation or reverse propagation, this option could be left open to the hypervisor. Along with each intercept bit that a hypervisor specifies in the VMCB/VMCS, the proposal is to define a redirect bit. For backward compatibility, the redirect bit could be specified as follows:

When the intercept bit is not set, the value in the redirect bit is ignored, and the hypervisor won’t get this intercept anyway. If the intercept bit is set but the corresponding redirect bit is cleared (which is the default case), then the hypervisor has priority over its guest in intercepting this event. If both bits are set, then the processor checks whether the guest is intercepting this event. If it does, then the intercept goes to the guest, otherwise it goes to the hypervisor.

5.3 The Hardware Algorithm

Thus the processor algorithm to determine which level of hypervisor to deliver an event is unified in pseudocode `intercept_delivery()`. The originating level is defined as the level where the currently executing code (pointed to by the instruction pointer `CS:RIP`) resides when the intercept occurs.

```c
intercept_delivery() {  
i := 0; j := 1  
while (j < originating level) and  
    ((level j intercept bit is 0) or  
     (level j redirect bit is 1)) {  
    if (level j intercepts) {  
        i := j  
    }  
    j := j + 1  
}  
if (j < originating level) or (i == 0) {  
    deliver the event to level j  
} else {  
    deliver the event to level i  
}
}
```

The algorithm always finds a definite level to deliver the event, so it itself will not generate double fault (#DF) or triple fault (SHUTDOWN) exceptions. A #DF or SHUTDOWN exception occurs only when a certain level has been selected to handle an event, and further faults occur while locating the corresponding handler in that same level.

For exceptions and interrupts (both internal and external), the top-level OS (as well as any hypervisor which is currently not running any guest, during which it behaves like a top-level OS) is poised to handle it anyway, as if it has the intercept bit set and the redirect bit cleared. So if each underlying hypervisor decides either not to handle an event or preferentially let its guest handle the event, then \( j \) equals the originating level at the end of the while-loop, and the top-level gets to handle the event.

For new intercepts that come only with the introduction of hypervisor (e.g. instruction intercept), the intercept event is generated only if at least one underlying hypervisor decides to intercept it. Hence \( j \) equals originating level implies \( i \neq 0 \) at the end of the while loop. The originating level will never get to receive the event (which it does not expect to receive).

If this algorithm is implemented in software, it would have the same linear running time (see section 5.4). It is not possible to achieve linear running time in software without paravirtualization because each forwarding step would incur more reverse propagation that avalanche down the hierarchy of hypervisors. Implementing this algorithm as a hardware extension maintains isolation between different levels of hypervisor.

5.4 Running Time Analysis

The running time is still dominated by the world switch cost \( t_{WS} \). From table 1, \( t_{WS} = 794 \) clock cycles. Now, hardware walks the ancestor and descendants linked lists to determine the correct level where intercept should be delivered. This walk is \( O(n) \), where \( n \) is the numerical value of the originating level. Thus the total running time is \( O(n + t_{WS}) \). This hardware algorithm is a great performance improvement (figure 4) to the exponential running time software solutions, and still keeps the hypervisors isolated from each other.

As the depth of nesting increases, the maximum delivery time increases linearly (instead of exponentially). The increase in variation is due to the flattening out of the minimum delivery time - delivery to the adjacent level takes constant time. The mathematical analysis closely matches the average case.
Figure 4: Propagation with hardware extension takes linear time.

6 Conclusion

There are many practical uses for recursive virtualization. VMExit forwarding is a key issue and performance bottleneck in building a hierarchical hypervisor. This paper proposed the concept of forward propagation and reverse propagation, and formulated a hypervisor-level distributed algorithm for its correct implementation, with its performance analyzed.

With the current x86 architecture, running time is exponential in the number of nesting levels. This exponential running time may be acceptable because the number of users to a hierarchical hypervisor also decreases exponentially with the number of nesting levels. However, the situation can be drastically improved to linear running time if a simple hardware extension as outlined in the last section is assumed. Such hardware extension could bring about more profound uses of recursive virtualization.

7 Acknowledgement

The first author wants to thank Benjamin Serebrin, Erich Boleyn and Kevin J. McGrath from AMD, as well as Ole Agesen and Jim Mattson from VMware for helping with the experiments reported herein and for improving the work.

References


[15] Wing-Chi Poon “Real-Time Hierarchical Hypervisor”, PhD Dissertation, August 2010, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, USA.


