The Turtles Project: Design and Implementation of Nested Virtualization

Muli Ben-Yehuda[†] Michael D. Day[‡] Zvi Dubitzky[†] Michael Factor[†] Nadav Har'El[†]
muli@il.ibm.com mdday@us.ibm.com dubi@il.ibm.com factor@il.ibm.com nyh@il.ibm.com
Abel Gordon[†] Anthony Liguori[‡] Orit Wasserman[†] Ben-Ami Yassour[†]
abelg@il.ibm.com aliguori@us.ibm.com oritw@il.ibm.com benami@il.ibm.com
[†]*IBM Research – Haifa* [‡]*IBM Linux Technology Center*

Abstract

In classical machine virtualization, a hypervisor runs multiple operating systems simultaneously, each on its own virtual machine. In *nested virtualization*, a hypervisor can run multiple other hypervisors with their associated virtual machines. As operating systems gain hypervisor functionality—Microsoft Windows 7 already runs Windows XP in a virtual machine—nested virtualization will become necessary in hypervisors that wish to host them. We present the design, implementation, analysis, and evaluation of high-performance nested virtualization on Intel x86-based systems. The Turtles project, which is part of the Linux/KVM hypervisor, runs multiple *unmodified* hypervisors (e.g., KVM and VMware) and operating systems (e.g., Linux and Windows). Despite the lack of architectural support for nested virtualization in the x86 architecture, it can achieve performance that is within 6-8% of single-level (non-nested) virtualization for common workloads, through *multi-dimensional paging* for MMU virtualization and *multi-level device assignment* for I/O virtualization.

> *The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever", said the old lady. "But it's turtles all the way down!"*[1]

1 Introduction

Commodity operating systems increasingly make use of virtualization capabilities in the hardware on which they run. Microsoft's newest operating system, Windows 7, supports a backward compatible Windows XP mode by running the XP operating system as a virtual machine. Linux has built-in hypervisor functionality via the KVM [29] hypervisor. As commodity operating systems gain virtualization functionality, nested virtualization will be required to run those operating systems/hypervisors themselves as virtual machines.

Nested virtualization has many other potential uses. Platforms with hypervisors embedded in firmware [1,20] need to support any workload and specifically other hypervisors as guest virtual machines. An Infrastructure-as-a-Service (IaaS) provider could give a user the ability to run a user-controlled hypervisor as a virtual machine. This way the cloud user could manage his own virtual machines directly with his favorite hypervisor of choice, and the cloud provider could attract users who would like to run their own hypervisors. Nested virtualization could also enable the live migration [14] of hypervisors and their guest virtual machines as a single entity for any reason, such as load balancing or disaster recovery. It also enables new approaches to computer security, such as honeypots capable of running hypervisor-level rootkits [43], hypervisor-level rootkit protection [39,44], and hypervisor-level intrusion detection [18, 25]—for both hypervisors and operating systems. Finally, it could also be used for testing, demonstrating, benchmarking and debugging hypervisors and virtualization setups.

The anticipated inclusion of nested virtualization in x86 operating systems and hypervisors raises many interesting questions, but chief amongst them is its runtime performance cost. Can it be made efficient enough that the overhead doesn't matter? We show that despite the lack of architectural support for nested virtualization in the x86 architecture, efficient nested x86 virtualization—with as little as 6-8% overhead—is feasible even when running *unmodified* binary-only hypervisors executing non-trivial workloads.

Because of the lack of architectural support for nested virtualization, an x86 guest hypervisor cannot use the hardware virtualization support directly to run its own guests. Fundamentally, our approach for nested virtualization *multiplexes* multiple levels of virtualization (mul-

[1] http://en.wikipedia.org/wiki/Turtles_all_the_way_down

tiple hypervisors) on the single level of architectural support available. We address each of the following areas: CPU (e.g., instruction-set) virtualization, memory (MMU) virtualization, and I/O virtualization.

x86 virtualization follows the "trap and emulate" model [21,22,36]. Since every trap by a guest hypervisor or operating system results in a trap to the lowest (most privileged) hypervisor, our approach for CPU virtualization works by having the lowest hypervisor inspect the trap and *forward* it to the hypervisors above it for emulation. We implement a number of optimizations to make world switches between different levels of the virtualization stack more efficient. For efficient memory virtualization, we developed *multi-dimensional paging*, which *collapses* the different memory translation tables into the one or two tables provided by the MMU [13]. For efficient I/O virtualization, we *bypass* multiple levels of hypervisor I/O stacks to provide nested guests with direct assignment of I/O devices [11, 31, 37, 52, 53] via *multi-level device assignment*.

Our main contributions in this work are:

- The design and implementation of nested virtualization for Intel x86-based systems. This implementation can run unmodified hypervisors such as KVM and VMware as guest hypervisors, and can run multiple operating systems such as Linux and Windows as nested virtual machines. Using multi-dimensional paging and multi-level device assignment, it can run common workloads with overhead as low as *6-8%* of single-level virtualization.

- The first evaluation and analysis of nested x86 virtualization performance, identifying the main causes of the virtualization overhead, and classifying them into guest hypervisor issues and limitations in the architectural virtualization support. We also suggest architectural and software-only changes which could reduce the overhead of nested x86 virtualization even further.

2 Related Work

Nested virtualization was first mentioned and theoretically analyzed by Popek and Goldberg [21, 22, 36]. Belpaire and Hsu extended this analysis and created a formal model [10]. Lauer and Wyeth [30] removed the need for a central supervisor and based nested virtualization on the ability to create nested virtual memories. Their implementation required hardware mechanisms and corresponding software support, which bear little resemblance to today's x86 architecture and operating systems.

Belpaire and Hsu also presented an alternative approach for nested virtualization [9]. In contrast to today's x86 architecture which has a single level of architectural support for virtualization, they proposed a hardware architecture with multiple virtualization levels.

The IBM z/VM hypervisor [35] included the first practical implementation of nested virtualization, by making use of multiple levels of architectural support. Nested virtualization was also implemented by Ford et al. in a microkernel setting [16] by modifying the software stack at all levels. Their goal was to enhance OS modularity, flexibility, and extensibility, rather than run unmodified hypervisors and their guests.

During the last decade software virtualization technologies for x86 systems rapidly emerged and were widely adopted by the market, causing both AMD and Intel to add virtualization extensions to their x86 platforms (AMD SVM [4] and Intel VMX [48]). KVM [29] was the first x86 hypervisor to support nested virtualization. Concurrent with this work, Alexander Graf and Joerg Roedel implemented nested support for AMD processors in KVM [23]. Despite the differences between VMX and SVM—VMX takes approximately twice as many lines of code to implement—nested SVM shares many of the same underlying principles as the Turtles project. Multi-dimensional paging was also added to nested SVM based on our work, but multi-level device assignment is not implemented.

There was also a recent effort to incorporate nested virtualization into the Xen hypervisor [24], which again appears to share many of the same underlying principles as our work. It is, however, at an early stage: it can only run a single nested guest on a single CPU, does not have multi-dimensional paging or multi-level device assignment, and no performance results have been published.

Blue Pill [43] is a root-kit based on hardware virtualization extensions. It is loaded during boot time by infecting the disk master boot record. It emulates VMX in order to remain functional and avoid detection when a hypervisor is installed in the system. Blue Pill's nested virtualization support is minimal since it only needs to remain undetectable [17]. In contrast, a hypervisor with nested virtualization support must efficiently multiplex the hardware across multiple levels of virtualization dealing with all of CPU, MMU, and I/O issues. Unfortunately, according to its creators, Blue Pill's nested VMX implementation can not be published.

ScaleMP vSMP is a commercial product which aggregates multiple x86 systems into a single SMP virtual machine. ScaleMP recently announced a new "VM on VM" feature which allows running a hypervisor on top of their underlying hypervisor. No details have been published on the implementation.

Berghmans demonstrates another approach to nested x86 virtualization, where a software-only hypervisor is run on a hardware-assisted hypervisor [12]. In contrast, our approach allows both hypervisors to take advantage of the virtualization hardware, leading to a more efficient implementation.

3 Turtles: Design and Implementation

The IBM Turtles nested virtualization project implements nested virtualization for Intel's virtualization technology based on the KVM [29] hypervisor. It can host multiple guest hypervisors simultaneously, each with its own multiple nested guest operating systems. We have tested it with unmodified KVM and VMware Server as guest hypervisors, and unmodified Linux and Windows as nested guest virtual machines. Since we treat nested hypervisors and virtual machines as unmodified black boxes, the Turtles project should also run any other x86 hypervisor and operating system.

The Turtles project is fairly mature: it has been tested running multiple hypervisors simultaneously, supports SMP, and takes advantage of two-dimensional page table hardware where available in order to implement nested MMU virtualization via multi-dimensional paging. It also makes use of multi-level device assignment for efficient nested I/O virtualization.

3.1 Theory of Operation

There are two possible models for nested virtualization, which differ in the amount of support provided by the underlying architecture. In the first model, *multi-level architectural support for nested virtualization*, each hypervisor handles all traps caused by sensitive instructions of any guest hypervisor running directly on top of it. This model is implemented for example in the IBM System z architecture [35].

The second model, *single-level architectural support for nested virtualization*, has only a single hypervisor mode, and a trap at any nesting level is handled by this hypervisor. As illustrated in Figure 1, regardless of the level in which a trap occurred, execution returns to the level 0 trap handler. Therefore, any trap occurring at any level from $1 \ldots n$ causes execution to drop to level 0. This limited model is implemented by both Intel and AMD in their respective x86 virtualization extensions, VMX [48] and SVM [4].

Since the Intel x86 architecture is a single-level virtualization architecture, only a single hypervisor can use the processor's VMX instructions to run its guests. For unmodified guest hypervisors to use VMX instructions, this single bare-metal hypervisor, which we call L_0, needs to emulate VMX. This emulation of VMX can work recursively. Given that L_0 provides a faithful emulation of the VMX hardware any time there is a trap on VMX instructions, the guest running on L_1 will not know it is not running directly on the hardware. Building on this infrastructure, the guest at L_1 is itself able use the same techniques to emulate the VMX hardware to an L_2 hypervisor which can then run its L_3 guests. More generally, given that the guest at L_{n-1} provides a faithful emulation of VMX to guests at L_n, a guest at L_n can use the exact same techniques to emulate VMX for a guest at L_{n+1}. We thus limit our discussion below to L_0, L_1, and L_2.

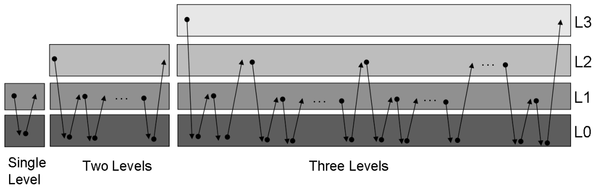

Figure 1: Nested traps with single-level architectural support for virtualization

Fundamentally, our approach for nested virtualization works by *multiplexing* multiple levels of virtualization (multiple hypervisors) on the single level of architectural support for virtualization, as can be seen in Figure 2. Traps are *forwarded* by L_0 between the different levels.

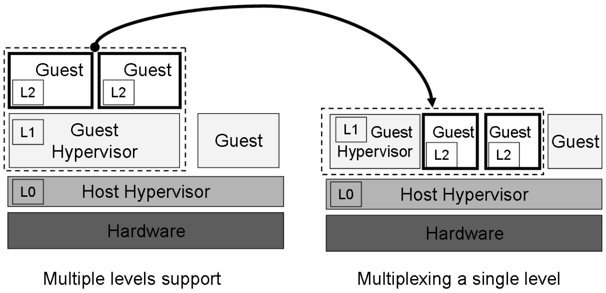

Figure 2: Multiplexing multiple levels of virtualization on a single hardware-provided level of support

When L_1 wishes to run a virtual machine, it launches it via the standard architectural mechanism. This causes a trap, since L_1 is not running in the highest privilege level (as is L_0). To run the virtual machine, L_1 supplies a specification of the virtual machine to be launched, which includes properties such as its initial instruction pointer and its page table root. This specification must be translated by L_0 into a specification that can be used to run L_2 directly on the bare metal, e.g., by converting memory addresses from L_1's physical address space to L_0's physical address space. Thus L_0 *multiplexes* the hardware between L_1 and L_2, both of which end up running as L_0 virtual machines.

When any hypervisor or virtual machine causes a trap, the L_0 trap handler is called. The trap handler then inspects the trapping instruction and its context, and decides whether that trap should be handled by L_0 (e.g., because the trapping context was L_1) or whether to forward it to the responsible hypervisor (e.g., because the trap occurred in L_2 and should be handled by L_1). In the latter case, L_0 forwards the trap to L_1 for handling.

When there are n levels of nesting guests, but the hardware supports less than n levels of MMU or DMA translation tables, the n levels need to be compressed onto the levels available in hardware, as described in Sections 3.3 and 3.4.

3.2 CPU: Nested VMX Virtualization

Virtualizing the x86 platform used to be complex and slow [40, 41, 49]. The hypervisor was forced to resort to on-the-fly binary translation of privileged instructions [3], slow machine emulation [8], or changes to guest operating systems at the source code level [6] or during compilation [32].

In due time Intel and AMD incorporated hardware virtualization extensions in their CPUs. These extensions introduced two new modes of operation: *root mode* and *guest mode*, enabling the CPU to differentiate between running a virtual machine (guest mode) and running the hypervisor (root mode). Both Intel and AMD also added special in-memory virtual machine control structures (VMCS and VMCB, respectively) which contain environment specifications for virtual machines and the hypervisor.

The VMX instruction set and the VMCS layout are explained in detail in [27]. Data stored in the VMCS can be divided into three groups. Guest state holds virtualized CPU registers (e.g., control registers or segment registers) which are automatically loaded by the CPU when switching from root mode to guest mode on VMEntry. Host state is used by the CPU to restore register values when switching back from guest mode to root mode on VMExit. Control data is used by the hypervisor to inject events such as exceptions or interrupts into virtual machines and to specify which events should cause a VMExit; it is also used by the CPU to specify the VMExit reason to the hypervisor.

In nested virtualization, the hypervisor running in root mode (L_0) runs other hypervisors (L_1) in guest mode. L_1 hypervisors have the illusion they are running in root mode. Their virtual machines (L_2) also run in guest mode.

As can be seen in Figure 3, L_0 is responsible for multiplexing the hardware between L_1 and L_2. The CPU runs L_1 using $\text{VMCS}_{0\rightarrow 1}$ environment specification. Respectively, $\text{VMCS}_{0\rightarrow 2}$ is used to run L_2. Both of these environment specifications are maintained by L_0. In addition, L_1 creates $\text{VMCS}_{1\rightarrow 2}$ within its own virtualized environment. Although $\text{VMCS}_{1\rightarrow 2}$ is never loaded into the processor, L_0 uses it to emulate a VMX enabled CPU for L_1.

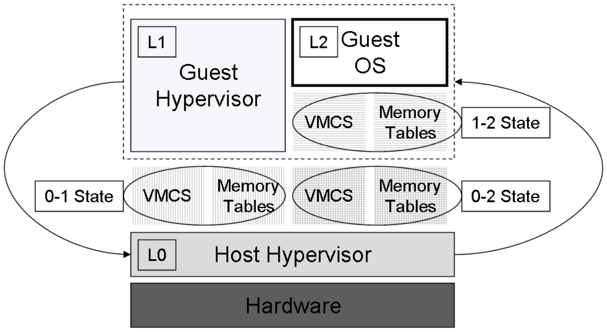

Figure 3: Extending VMX for nested virtualization

3.2.1 VMX Trap and Emulate

VMX instructions can only execute successfully in root mode. In the nested case, L_1 uses VMX instructions in guest mode to load and launch L_2 guests, which causes VMExits. This enables L_0, running in root mode, to trap and emulate the VMX instructions executed by L_1.

In general, when L_0 emulates VMX instructions, it updates VMCS structures according to the update process described in the next section. Then, L_0 resumes L_1, as though the instructions were executed directly by the CPU. Most of the VMX instructions executed by L_1 cause, first, a VMExit from L_1 to L_0, and then a VMEntry from L_0 to L_1.

For the instructions used to run a new VM, `vmresume` and `vmlaunch`, the process is different, since L_0 needs to emulate a VMEntry from L_1 to L_2. Therefore, any execution of these instructions by L_1 cause, first, a VMExit from L_1 to L_0, and then, a VMEntry from L_0 to L_2.

3.2.2 VMCS Shadowing

L_0 prepares a VMCS ($\text{VMCS}_{0\rightarrow 1}$) to run L_1, exactly in the same way a hypervisor executes a guest with a single level of virtualization. From the hardware's perspective, the processor is running a single hypervisor (L_0) in root mode and a guest (L_1) in guest mode. L_1 is not aware that it is running in guest mode and uses VMX instructions to create the specifications for its own guest, L_2.

L_1 defines L_2's environment by creating a VMCS ($\text{VMCS}_{1\rightarrow 2}$) which contains L_2's environment from L_1's perspective. For example, the $\text{VMCS}_{1\rightarrow 2}$ GUEST-CR3 field points to the page tables that L_1 prepared for L_2. L_0 cannot use $\text{VMCS}_{1\rightarrow 2}$ to execute L_2 directly, since $\text{VMCS}_{1\rightarrow 2}$ is not valid in L_0's environment and L_0 cannot use L_1's page tables to run L_2. Instead, L_0 uses

VMCS$_{1\to2}$ to construct a new VMCS (VMCS$_{0\to2}$) that holds L$_2$'s environment from L$_0$'s perspective.

L$_0$ must consider all the specifications defined in VMCS$_{1\to2}$ and also the specifications defined in VMCS$_{0\to1}$ to create VMCS$_{0\to2}$. The host state defined in VMCS$_{0\to2}$ must contain the values required by the CPU to correctly switch back from L$_2$ to L$_0$. In addition, VMCS$_{1\to2}$ host state must be copied to VMCS$_{0\to1}$ guest state. Thus, when L$_0$ emulates a switch between L$_2$ to L$_1$, the processor loads the correct L$_1$ specifications.

The guest state stored in VMCS$_{1\to2}$ does not require any special handling in general, and most fields can be copied directly to the guest state of VMCS$_{0\to2}$.

The control data of VMCS$_{1\to2}$ and VMCS$_{0\to1}$ must be merged to correctly emulate the processor behavior. For example, consider the case where L$_1$ specifies to trap an event E_A in VMCS$_{1\to2}$ but L$_0$ does not trap such event for L$_1$ (i.e., a trap is not specified in VMCS$_{0\to1}$). To forward the event E_A to L$_1$, L$_0$ needs to specify the corresponding trap in VMCS$_{0\to2}$. In addition, the field used by L$_1$ to inject events to L$_2$ needs to be merged, as well as the fields used by the processor to specify the exit cause.

For the sake of brevity, we omit some details on how specific VMCS fields are merged. For the complete details, the interested reader is encouraged to refer to the KVM source code [29].

3.2.3 VMEntry and VMExit Emulation

In nested environments, switches from L$_1$ to L$_2$ and back must be emulated. When L$_2$ is running and a VMExit occurs there are two possible handling paths, depending on whether the VMExit must be handled only by L$_0$ or must be forwarded to L$_1$.

When the event causing the VMExit is related to L$_0$ only, L$_0$ handles the event and resumes L$_2$. This kind of event can be an external interrupt, a non-maskable interrupt (NMI) or any trappable event specified in VMCS$_{0\to2}$ that was not specified in VMCS$_{1\to2}$. From L$_1$'s perspective this event does not exist because it was generated outside the scope of L$_1$'s virtualized environment. By analogy to the non-nested scenario, an event occurred at the hardware level, the CPU transparently handled it, and the hypervisor continued running as before.

The second handling path is caused by events related to L$_1$ (e.g., trappable events specified in VMCS$_{1\to2}$). In this case L$_0$ forwards the event to L$_1$ by copying VMCS$_{0\to2}$ fields updated by the processor to VMCS$_{1\to2}$ and resuming L$_1$. The hypervisor running in L$_1$ believes there was a VMExit directly from L$_2$ to L$_1$. The L$_1$ hypervisor handles the event and later on resumes L$_2$ by executing `vmresume` or `vmlaunch`, both of which will be emulated by L$_0$.

3.3 MMU: Multi-dimensional Paging

In addition to virtualizing the CPU, a hypervisor also needs to virtualize the MMU: A guest OS builds a guest page table which translates guest virtual addresses to guest physical addresses. These must be translated again into host physical addresses. With nested virtualization, a third layer of address translation is needed.

These translations can be done entirely in software, or assisted by hardware. However, as we explain below, current hardware supports only one or two *dimensions* (levels) of translation, not the three needed for nested virtualization. In this section we present a new technique, *multi-dimensional paging*, for multiplexing the three needed translation tables onto the two available in hardware. In Section 4.1.2 we demonstrate the importance of this technique, showing that more naïve approaches (surveyed below) cause at least a three-fold slowdown of some useful workloads.

When no hardware support for memory management virtualization was available, a technique known as shadow page tables [15] was used. A guest creates a guest page table, which translates guest virtual addresses to guest physical addresses. Based on this table, the hypervisor creates a new page table, the *shadow page table*, which translates guest virtual addresses directly to the corresponding host physical address [3, 6]. The hypervisor then runs the guest using this shadow page table instead of the guest's page table. The hypervisor has to trap all guest paging changes, including page fault exceptions, the INVLPG instruction, context switches (which cause the use of a different page table) and all the guest updates to the page table.

To improve virtualization performance, x86 architectures recently added *two-dimensional page tables* [13]—a second translation table in the hardware MMU. When translating a guest virtual address, the processor first uses the regular guest page table to translate it to a guest physical address. It then uses the second table, called EPT by Intel (and NPT by AMD), to translate the guest physical address to a host physical address. When an entry is missing in the EPT table, the processor generates an EPT violation exception. The hypervisor is responsible for maintaining the EPT table and its cache (which can be flushed with INVEPT), and for handling EPT violations, while guest page faults can be handled entirely by the guest.

The hypervisor, depending on the processors capabilities, decides whether to use shadow page tables or two-dimensional page tables to virtualize the MMU. In nested environments, both hypervisors, L$_0$ and L$_1$, determine independently the preferred mechanism. Thus, L$_0$ and L$_1$ hypervisors can use the same or a different MMU virtualization mechanism. Figure 4 shows three different nested MMU virtualization models.

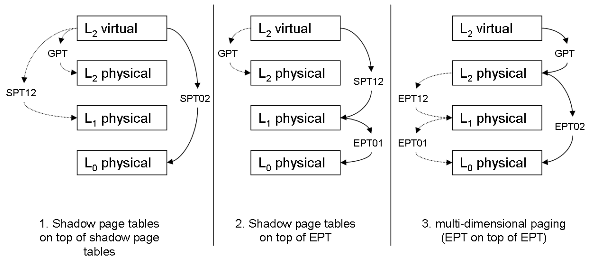

Figure 4: MMU alternatives for nested virtualization

Shadow-on-shadow is used when the processor does not support two-dimensional page tables, and is the least efficient method. Initially, L_0 creates a shadow page table to run L_1 ($\mathrm{SPT}_{0\rightarrow 1}$). L_1, in turn, creates a shadow page table to run L_2 ($\mathrm{SPT}_{1\rightarrow 2}$). L_0 cannot use $\mathrm{SPT}_{1\rightarrow 2}$ to run L_2 because this table translates L_2 guest virtual addresses to L_1 host physical addresses. Therefore, L_0 compresses $\mathrm{SPT}_{0\rightarrow 1}$ and $\mathrm{SPT}_{1\rightarrow 2}$ into a single shadow page table, $\mathrm{SPT}_{0\rightarrow 2}$. This new table translates directly from L_2 guest virtual addresses to L_0 host physical addresses. Specifically, for each guest virtual address in $\mathrm{SPT}_{1\rightarrow 2}$, L_0 creates an entry in $\mathrm{SPT}_{0\rightarrow 2}$ with the corresponding L_0 host physical address.

Shadow-on-EPT is the most straightforward approach to use when the processor supports EPT. L_0 uses the EPT hardware, but L_1 cannot use it, so it resorts to shadow page tables. L_1 uses $\mathrm{SPT}_{1\rightarrow 2}$ to run L_2. L_0 configures the MMU to use $\mathrm{SPT}_{1\rightarrow 2}$ as the first translation table and $\mathrm{EPT}_{0\rightarrow 1}$ as the second translation table. In this way, the processor first translates from L_2 guest virtual address to L_1 host physical address using $\mathrm{SPT}_{1\rightarrow 2}$, and then translates from the L_1 host physical address to the L_0 host physical address using the $\mathrm{EPT}_{0\rightarrow 1}$.

Though the Shadow-on-EPT approach uses the EPT hardware, it still has a noticeable overhead due to page faults and page table modifications in L_2. These must be handled in L_1, to maintain the shadow page table. Each of these faults and writes cause VMExits and must be forwarded from L_0 to L_1 for handling. In other words, Shadow-on-EPT is slow for the exactly the same reasons that Shadow itself was slow for single-level virtualization—but it is even slower because nested exits are slower than non-nested exits.

In *multi-dimensional page tables*, as in two-dimensional page tables, each level creates its own separate translation table. For L_1 to create an EPT table, L_0 exposes EPT capabilities to L_1, even though the hardware only provides a single EPT table.

Since only one EPT table is available in hardware, the two EPT tables should be *compressed* into one: Let us assume that L_0 runs L_1 using $\mathrm{EPT}_{0\rightarrow 1}$, and that L_1 creates an additional table, $\mathrm{EPT}_{1\rightarrow 2}$, to run L_2, because L_0 exposed a virtualized EPT capability to L_1. The L_0 hypervisor could then compress $\mathrm{EPT}_{0\rightarrow 1}$ and $\mathrm{EPT}_{1\rightarrow 2}$ into a single $\mathrm{EPT}_{0\rightarrow 2}$ table as shown in Figure 4. Then L_0 could run L_2 using $\mathrm{EPT}_{0\rightarrow 2}$, which translates directly from the L_2 guest physical address to the L_0 host physical address, reducing the number of page fault exits and improving nested virtualization performance. In Section 4.1.2 we demonstrate more than a three-fold speedup of some useful workloads with multi-dimensional page tables, compared to shadow-on-EPT.

The L_0 hypervisor launches L_2 with an empty $\mathrm{EPT}_{0\rightarrow 2}$ table, building the table on-the-fly, on L_2 EPT-violation exits. These happen when a translation for a guest physical address is missing in the EPT table. If there is no translation in $\mathrm{EPT}_{1\rightarrow 2}$ for the faulting address, L_0 first lets L_1 handle the exit and update $\mathrm{EPT}_{1\rightarrow 2}$. L_0 can now create an entry in $\mathrm{EPT}_{0\rightarrow 2}$ that translates the L_2 guest physical address directly to the L_0 host physical address: $\mathrm{EPT}_{1\rightarrow 2}$ is used to translate the L_2 physical address to a L_1 physical address, and $\mathrm{EPT}_{0\rightarrow 1}$ translates that into the desired L_0 physical address.

To maintain correctness of $\mathrm{EPT}_{0\rightarrow 2}$, the L_0 hypervisor needs to know of any changes that L_1 makes to $\mathrm{EPT}_{1\rightarrow 2}$. L_0 sets the memory area of $\mathrm{EPT}_{1\rightarrow 2}$ as read-only, thereby causing a trap when L_1 tries to update it. L_0 will then update $\mathrm{EPT}_{0\rightarrow 2}$ according to the changed entries in $\mathrm{EPT}_{1\rightarrow 2}$. L_0 also needs to trap all L_1 INVEPT instructions, and invalidate the EPT cache accordingly.

By using *huge pages* [34] to back guest memory, L_0 can create smaller and faster EPT tables. Finally, to further improve performance, L_0 also allows L_1 to use *VPID*s. With this feature, the CPU tags each translation in the TLB with a numeric *virtual-processor id*, eliminating the need for TLB flushes on every VMEntry and VMExit. Since each hypervisor is free to choose these VPIDs arbitrarily, they might collide and therefore L_0 needs to map the VPIDs that L_1 uses into valid L_0 VPIDs.

3.4 I/O: Multi-level Device Assignment

I/O is the third major challenge in server virtualization. There are three approaches commonly used to provide I/O services to a guest virtual machine. Either the hypervisor *emulates* a known device and the guest uses an unmodified driver to interact with it [47], or a *para-virtual driver* is installed in the guest [6,42], or the host assigns a real device to the guest which then controls the device directly [11,31,37,52,53]. Device assignment generally provides the best performance [33,38,53], since it minimizes the number of I/O-related world switches between the virtual machine and its hypervisor, and although it complicates live migration, device assignment and live migration can peacefully coexist [26, 28, 54].

These three basic I/O approaches for a single-level guest imply nine possible combinations in the two-level nested guest case. Of the nine potential combinations we evaluated the more interesting cases, presented in Table 1. Implementing the first four alternatives is straightforward. We describe the last option, which we call *multi-level device assignment*, below. Multi-level device assignment lets the L_2 guest access a device directly, bypassing both hypervisors. This direct device access requires dealing with DMA, interrupts, MMIO, and PIOs [53].

I/O virtualization method between L_0 & L_1	I/O virtualization method between L_1 & L_2
Emulation	Emulation
Para-virtual	Emulation
Para-virtual	Para-virtual
Device assignment	Para-virtual
Device assignment	Device assignment

Table 1: I/O combinations for a nested guest

Device DMA in virtualized environments is complicated, because guest drivers use guest physical addresses, while memory access in the device is done with host physical addresses. The common solution to the DMA problem is an IOMMU [2, 11], a hardware component which resides between the device and main memory. It uses a translation table prepared by the hypervisor to translate the guest physical addresses to host physical addresses. IOMMUs currently available, however, only support a single level of address translation. Again, we need to compress two levels of translation tables onto the one level available in hardware.

For modified guests this can be done using a paravirtual IOMMU: the code in L_1 which sets a mapping on the IOMMU from L_2 to L_1 addresses is replaced by a hypercall to L_0. L_0 changes the L_1 address in that mapping to the respective L_0 address, and puts the resulting mapping (from L_2 to L_0 addresses) in the IOMMU.

A better approach, one which can run unmodified guests, is for L_0 to emulate an IOMMU for L_1 [5]. L_1 believes that it is running on a machine with an IOMMU, and sets up mappings from L_2 to L_1 addresses on it. L_0 intercepts these mappings, remaps the L_1 addresses to L_0 addresses, and builds the L_2-to-L_0 map on the real IOMMU.

In current x86 architecture, interrupts always cause a guest exit to L_0, which proceeds to forward the interrupt to L_1. L_1 will then inject it into L_2. The EOI (end of interrupt) will also cause a guest exit. In Section 4.1.1 we discuss the slowdown caused by these interrupt-related exits, and propose ways to avoid it.

Memory-mapped I/O (MMIO) and Port I/O (PIO) for a nested guest work the same way they work for a single-level guest, without incurring exits on the critical I/O path [53].

3.5 Micro Optimizations

There are two main places where a guest of a nested hypervisor is slower than the same guest running on a bare-metal hypervisor. First, the transitions between L_1 and L_2 are slower than the transitions between L_0 and L_1. Second, the exit handling code running in the L_1 hypervisor is slower than the same code running in L_0. In this section we discuss these two issues, and propose optimizations that improve performance. Since we assume that both L_1 and L_2 are unmodified, these optimizations require modifying L_0 only. We evaluate these optimizations in the evaluation section.

3.5.1 Optimizing transitions between L_1 and L_2

As explained in Section 3.2.3, transitions between L_1 and L_2 involve an exit to L_0 and then an entry. In L_0, most of the time is spent merging the VMCS's. We optimize this merging code to only copy data between VMCS's if the relevant values were modified. Keeping track of which values were modified has an intrinsic cost, so one must carefully balance full copying versus partial copying and tracking. We observed empirically that for common workloads and hypervisors, partial copying has a lower overhead.

VMCS merging could be further optimized by copying multiple VMCS fields at once. However, according to Intel's specifications, reads or writes to the VMCS area must be performed using `vmread` and `vmwrite` instructions, which operate on a single field. We empirically noted that under certain conditions one could access VMCS data directly without ill side-effects, bypassing `vmread` and `vmwrite` and copying multiple fields at once with large memory copies. However, this optimization does not strictly adhere to the VMX specifications, and thus might not work on processors other than the ones we have tested.

In the evaluation section, we show that this optimization gives a significant performance boost in micro-benchmarks. However, it did not noticeably improve the other, more typical, workloads that we have evaluated.

3.5.2 Optimizing exit handling in L_1

The exit-handling code in the hypervisor is slower when run in L_1 than the same code running in L_0. The main cause of this slowdown are additional exits caused by privileged instructions in the exit-handling code.

In Intel VMX, the privileged instructions `vmread` and `vmwrite` are used by the hypervisor to read and modify the guest and host specification. As can be seen in Section 4.3, these cause L_1 to exit multiple times while it handles a single L_2 exit.

In contrast, in AMD SVM, guest and host specifications can be read or written to directly using ordinary memory loads and stores. The clear advantage of that model is that L_0 does not intervene while L_1 modifies L_2 specifications. Removing the need to trap and emulate special instructions reduces the number of exits and improves nested virtualization performance.

One thing L_0 can do to avoid trapping on every `vmread` and `vmwrite` is binary translation [3] of problematic `vmread` and `vmwrite` instructions in the L_1 instruction stream, by trapping the first time such an instruction is called and then rewriting it to branch to a non-trapping memory load or store. To evaluate the potential performance benefit of this approach, we tested a modified L_1 that directly reads and writes VMCS$_{1\rightarrow 2}$ in memory, instead of using `vmread` and `vmwrite`. The performance of this setup, which we call *DRW* (direct read and write) is described in the evaluation section.

4 Evaluation

We start the evaluation and analysis of nested virtualization with macro benchmarks that represent real-life workloads. Next, we evaluate the contribution of multi-level device assignment and multi-dimensional paging to nested virtualization performance. Most of our experiments are executed with KVM as the L_1 guest hypervisor. In Section 4.2 we present results with VMware Server as the L_1 guest hypervisor.

We then continue the evaluation with a synthetic, worst-case micro benchmark running on L_2 which causes guest exits in a loop. We use this synthetic, worst-case benchmark to understand and analyze the overhead and the handling flow of a single L_2 exit.

Our setup consisted of an IBM x3650 machine booted with a single Intel Xeon 2.9GHz core and with 3GB of memory. The host OS was Ubuntu 9.04 with a kernel that is based on the KVM git tree version kvm-87, with our nested virtualization support added. For both L_1 and L_2 guests we used an Ubuntu Jaunty guest with a kernel that is based on the KVM git tree, version kvm-87. L_1 was configured with 2GB of memory and L_2 was configured with 1GB of memory. For the I/O experiments we used a Broadcom NetXtreme 1Gb/s NIC connected via crossover-cable to an e1000e NIC on another machine.

4.1 Macro Workloads

`kernbench` is a general purpose compilation-type benchmark that compiles the Linux kernel multiple times. The compilation process is, by nature, CPU- and memory-intensive, and it also generates disk I/O to load the compiled files into the guest's page cache.

`SPECjbb` is an industry-standard benchmark designed to measure the server-side performance of Java run-time environments. It emulates a three-tier system and is primarily CPU-intensive.

We executed `kernbench` and `SPECjbb` in four setups: host, single-level guest, nested guest, and nested guest optimized with direct read and write (DRW) as described in Section 3.5.2. The optimizations described in Section 3.5.1 did not make a significant difference to these benchmarks, and are thus omitted from the results. We used KVM as both L_0 and L_1 hypervisor with multi-dimensional paging. The results are depicted in Table 2.

Kernbench				
	Host	Guest	Nested	Nested$_{DRW}$
Run time	324.3	355	406.3	391.5
STD dev.	1.5	10	6.7	3.1
% overhead vs. host	-	9.5	25.3	20.7
% overhead vs. guest	-	-	14.5	10.3
%CPU	93	97	99	99

SPECjbb				
	Host	Guest	Nested	Nested$_{DRW}$
Score	90493	83599	77065	78347
STD dev.	1104	1230	1716	566
% degradation vs. host	-	7.6	14.8	13.4
% degradation vs. guest	-	-	7.8	6.3
%CPU	100	100	100	100

Table 2: `kernbench` and `SPECjbb` results

We compared the impact of running the workloads in a nested guest with running the same workload in a single-level guest, i.e., the overhead added by the additional level of virtualization. For `kernbench`, the overhead of nested virtualization is 14.5%, while for SPECjbb the score is degraded by 7.82%. When we discount the Intel-specific `vmread` and `vmwrite` overhead in L_1, the overhead is 10.3% and 6.3% respectively.

To analyze the sources of overhead, we examine the time distribution between the different levels. Figure 5 shows the time spent in each level. It is interesting to compare the time spent in the hypervisor in the single-level case with the time spent in L_1 in the nested guest case, since both hypervisors are expected to do the same work. The times are indeed similar, although the L_1 hypervisor takes more cycles due to cache pollution and TLB flushes, as we show in Section 4.3. The significant part of the virtualization overhead in the nested case comes from the time spent in L_0 and the increased number of exits.

For SPECjbb, the total number of cycles across all levels is the same for all setups. This is because SPECjbb executed for the same pre-set amount of time in both cases and the difference was in the benchmark score.

Efficiently virtualizing a hypervisor is hard. Nested virtualization creates a new kind of workload for the L_0 hypervisor which did not exist before: running another hypervisor (L_1) as a guest. As can be seen in Figure 5, for kernbench L_0 takes only 2.28% of the overall cycles in the single-level guest case, but takes 5.17% of the overall cycles for the nested-guest case. In other words, L_0 has to work more than twice as hard when running a nested guest.

Not all exits of L_2 incur the same overhead, as each type of exit requires different handling in L_0 and L_1. In Figure 6, we show the total number of cycles required to handle each exit type. For the single level guest we measured the number of cycles between VMExit and the consequent VMEntry. For the nested guest we measured the number of cycles spent between L_2 VMExit and the consequent L_2 VMEntry.

There is a large variance between the handling times of different types of exits. The cost of each exit comes primarily from the number of privileged instructions performed by L_1, each of which causes an exit to L_0. For example, when L_1 handles a PIO exit of L_2, it generates on average 31 additional exits, whereas in the cpuid case discussed later in Section 4.3 only 13 exits are required. Discounting traps due to vmread and vmwrite, the average number of exits was reduced to 14 for PIO and to 2 for cpuid.

Another source of overhead is heavy-weight exits. The external interrupt exit handler takes approximately 64K cycles when executed by L_0. The PIO exit handler takes approximately 12K cycles when executed by L_0. However, when those handlers are executed by L_1, they take much longer: approximately 192K cycles and 183K cycles, respectively. Discounting traps due to vmread and vmwrite, they take approximately 148K cycles and 130K cycles, respectively. This difference in execution times between L_0 and L_1 is due to two reasons: first, the handlers execute privileged instructions causing exits to L_0. Second, the handlers run for a long time compared with other handlers and therefore more external events such as external interrupts occur during their run-time.

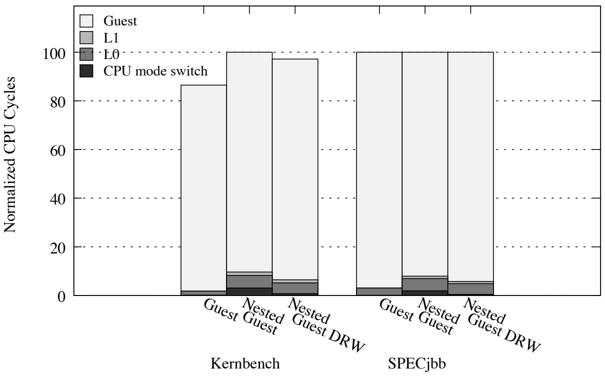

Figure 5: CPU cycle distribution

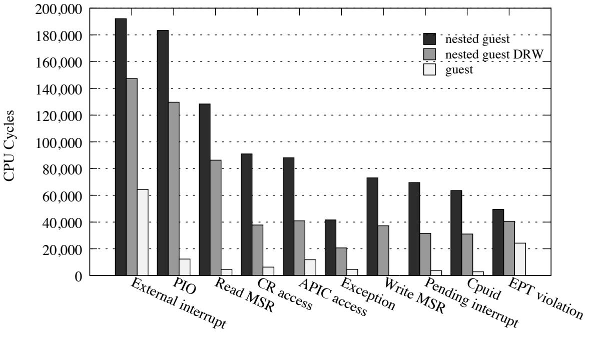

Figure 6: Cycle costs of handling different types of exits

4.1.1 I/O Intensive Workloads

To examine the performance of a nested guest in the case of I/O intensive workloads we used netperf, a TCP streaming application that attempts to maximize the amount of data sent over a single TCP connection. We measured the performance on the sender side, with the default settings of netperf (16,384 byte messages).

Figure 7 shows the results for running the netperf TCP stream test on the host, in a single-level guest, and in a nested guest, using the five I/O virtualization combinations described in Section 3.4. We used KVM's default emulated NIC (RTL-8139), virtio [42] for a paravirtual NIC, and a 1 Gb/s Broadcom NetXtreme II with device assignment. All tests used a single CPU core.

On bare-metal, netperf easily achieved line rate (940 Mb/s) with 20% CPU utilization.

Emulation gives a much lower throughput, with full CPU utilization: On a single-level guest we get 25% of the line rate. On the nested guest the throughput is even lower and the overhead is dominated by the cost of device emulation between L_1 and L_2. Each L_2 exit is trapped by L_0 and forwarded to L_1. For each L_2 exit, L_1 then executes multiple privileged instructions, incurring multiple exits back to L_0. In this way the overhead for

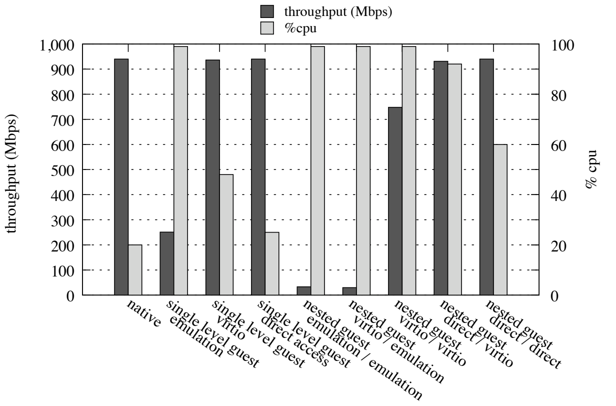

Figure 7: Performance of `netperf` in various setups

each L_2 exit is multiplied.

The para-virtual virtio NIC performs better than emulation since it reduces the number of exits. Using virtio all the way up to L_2 gives 75% of line rate with a saturated CPU, better but still considerably below bare-metal performance.

Multi-level device assignment achieved the best performance, with line rate at 60% CPU utilization (Figure 7, *direct/direct*). Using device assignment between L_0 and L_1 and virtio between L_1 and L_2 enables the L_2 guest to saturate the 1Gb link with 92% CPU utilization (Figure 7, *direct/virtio*).

While multi-level device assignment outperformed the other methods, its measured performance is still suboptimal because 60% of the CPU is used for running a workload that only takes 20% on bare-metal. Unfortunately on current x86 architecture, interrupts cannot be assigned to guests, so both the interrupt itself and its EOI cause exits. The more interrupts the device generates, the more exits, and therefore the higher the virtualization overhead—which is more pronounced in the nested case. We hypothesize that these interrupt-related exits are the biggest source of the remaining overhead, so had the architecture given us a way to avoid these exits—by assigning interrupts directly to guests rather than having each interrupt go through both hypervisors—`netperf` performance on L_2 would be close to that of bare-metal.

To test this hypothesis we reduced the number of interrupts, by modifying standard *bnx2* network driver to work without any interrupts, i.e., continuously poll the device for pending events

Figure 8 compares some of the I/O virtualization combinations with this polling driver. Again, multi-level device assignment is the best option and, as we hypothesized, this time L_2 performance is close to bare-metal. With `netperf`'s default 16,384 byte messages, the throughput is often capped by the 1 Gb/s line rate, so we ran netperf with smaller messages. As we can see in the figure, for 64-byte messages, for example, on L_0 (bare metal) a throughput of 900 Mb/s is achieved, while on L_2 with multi-level device assignment, we get 837 Mb/s, a mere 7% slowdown. The runner-up method, virtio on direct, was not nearly as successful, and achieved just 469 Mb/s, 50% below bare-metal performance. CPU utilization was 100% in all cases since a polling driver consumes all available CPU cycles.

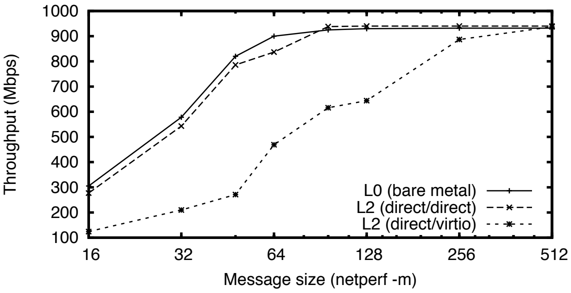

Figure 8: Performance of `netperf` with interrupt-less network driver

4.1.2 Impact of Multi-dimensional Paging

To evaluate multi-dimensional paging, we compared each of the macro benchmarks described in the previous sections with and without multi-dimensional paging. For each benchmark we configured L_0 to run L_1 with EPT support. We then compared the case where L_1 uses shadow page tables to run L_2 ("Shadow-on-EPT") with the case of L_1 using EPT to run L_2 ("multi-dimensional paging").

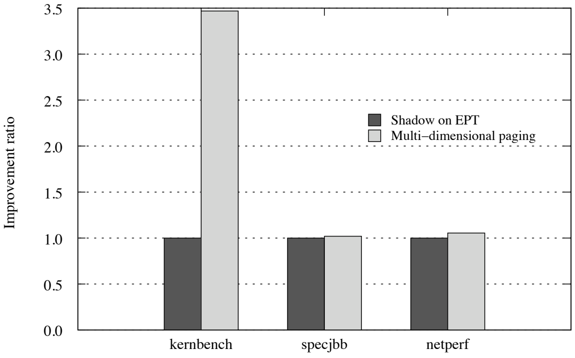

Figure 9: Impact of multi-dimensional paging

Figure 9 shows the results. The overhead between the two cases is mostly due to the number of page-fault exits. When shadow paging is used, each page fault of the L_2 guest results in a VMExit. When multi-dimensional paging is used, only an access to a guest physical page that is not mapped in the EPT table will cause an EPT violation exit. Therefore the impact of multi-dimensional paging depends on the number of guest page faults, which is a property of the workload. The improvement is startling in benchmarks such as kernbench with a high number of page faults, and is less pronounced in workloads that do not incur many page faults.

4.2 VMware Server as a Guest Hypervisor

We also evaluated VMware as the L_1 hypervisor to analyze how a different guest hypervisor affects nested virtualization performance. We used the hosted version, VMWare Server v2.0.1, build 156745 x86-64, on top of Ubuntu based on kernel 2.6.28-11. We intentionally did not install VMware tools for the L_2 guest, thereby increasing nested virtualization overhead. Due to similar results obtained for VMware and KVM as the nested hypervisor, we show only kernbench and SPECjbb results below.

Benchmark	% overhead vs. single-level guest
kernbench	14.98
SPECjbb	8.85

Table 3: VMware Server as a guest hypervisor

Examining L_1 exits, we noticed VMware Server uses VMX initialization instructions (vmon, vmoff, vmptrld, vmclear) several times during L_2 execution. Conversely, KVM uses them only once. This dissimilitude derives mainly from the approach used by VMware to interact with the host Linux kernel. Each time the monitor module takes control of the CPU, it enables VMX. Then, before it releases control to the Linux kernel, VMX is disabled. Furthermore, during this transition many non-VMX privileged instructions are executed by L_1, increasing L_0 intervention.

Although all these initialization instructions are emulated by L_0, transitions from the VMware monitor module to the Linux kernel are less frequent for Kernbench and SPECjbb. The VMware monitor module typically handles multiple L_2 exits before switching to the Linux kernel. As a result, this behavior only slightly affected the nested virtualization performance.

4.3 Micro Benchmark Analysis

To analyze the cycle-costs of handling a single L_2 exit, we ran a micro benchmark in L_2 that does nothing except generate exits by calling cpuid in a loop. The virtualization overhead for running an L_2 guest is the ratio between the effective work done by the L_2 guest and the overhead of handling guest exits in L_0 and L_1. Based on this definition, this cpuid micro benchmark is a worst case workload, since L_2 does virtually nothing except generate exits. We note that cpuid cannot in the general case be handled by L_0 directly, as L_1 may wish to modify the values returned to L_2.

Figure 10 shows the number of CPU cycles required to execute a single cpuid instruction. We ran the cpuid instruction $4*10^6$ times and calculated the average number of cycles per iteration. We repeated the test for the following setups: 1. native, 2. running cpuid in a single level guest, and 3. running cpuid in a nested guest with and without the optimizations described in Section 3.5. For each execution, we present the distribution of the cycles between the levels: L_0, L_1, L_2. CPU mode switch stands for the number of cycles spent by the CPU when performing a VMEntry or a VMExit. On bare metal cpuid takes about 100 cycles, while in a virtual machine it takes about 2,600 cycles (Figure 10, *column 1*), about 1,000 of which is due to the CPU mode switching. When run in a nested virtual machine it takes about 58,000 cycles (Figure 10, *column 2*).

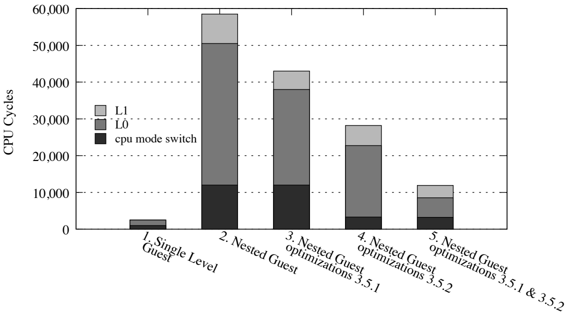

Figure 10: CPU cycle distribution for cpuid

To understand the cost of handling a nested guest exit compared to the cost of handling the same exit for a single-level guest, we analyzed the flow of handling cpuid:

1. L_2 executes a cpuid instruction
2. CPU traps and switches to root mode L_0
3. L_0 switches state from running L_2 to running L_1
4. CPU switches to guest mode L_1
5. L_1 modifies $\text{VMCS}_{1\to2}$
   repeat n times:
   (a) L_1 accesses $\text{VMCS}_{1\to2}$
   (b) CPU traps and switches to root mode L_0
   (c) L_0 emulates $\text{VMCS}_{1\to2}$ access and resumes L_1
   (d) CPU switches to guest mode L_1
6. L_1 emulates cpuid for L_2

7. L_1 executes a resume of L_2
8. CPU traps and switches to root mode L_0
9. L_0 switches state from running L_1 to running L_2
10. CPU switches to guest mode L_2

In general, step 5 can be repeated multiple times. Each iteration consists of a single VMExit from L_1 to L_0. The total number of exits depends on the specific implementation of the L_1 hypervisor. A nesting-friendly hypervisor will keep privileged instructions to a minimum. In any case, the L_1 hypervisor must interact with VMCS$_{1\to2}$, as described in Section 3.2.2. In the case of `cpuid`, in step 5, L_1 reads 7 fields of VMCS$_{1\to2}$, and writes 4 fields to VMCS$_{1\to2}$, which ends up as 11 VMExits from L_1 to L_0. Overall, for a single L_2 `cpuid` exit there are 13 CPU mode switches from guest mode to root mode and 13 CPU mode switches from root mode to guest mode, specifically in steps: 2, 4, 5b, 5d, 8, 10.

The number of cycles the CPU spends in a single switch to guest mode plus the number of cycles to switch back to root mode, is approximately 1,000. The total CPU switching cost is therefore around 13,000 cycles.

The other two expensive steps are 3 and 9. As described in Section 3.5, these switches can be optimized. Indeed as we show in Figure 10, *column 3*, using various optimizations we can reduce the virtualization overhead by 25%, and by 80% when using non-trapping `vmread` and `vmwrite` instructions.

By avoiding traps on `vmread` and `vmwrite` (*Figure 10, columns 4 and 5*), we removed the exits caused by VMCS$_{1\to2}$ accesses and the corresponding VMCS access emulation, step 5. This optimization reduced the switching cost by 84.6%, from 13,000 to 2,000.

While it might still be possible to optimize steps 3 and 9 further, it is clear that the exits of L_1 while handling a single exit of L_2, and specifically VMCS accesses, are a major source of overhead. Architectural support for both faster world switches and VMCS updates without exits will reduce the overhead.

Examining Figure 10, it seems that handling `cpuid` in L_1 is more expensive than handling `cpuid` in L_0. Specifically, in column 3, the nested hypervisor L_1 spends around 5,000 cycles to handle `cpuid`, while in column 1 the same hypervisor running on bare metal only spends 1500 cycles to handle the same exit (note that these numbers do not include the mode switches). The code running in L_1 and in L_0 is identical; the difference in cycle count is due to cache pollution. Running the `cpuid` handling code incurs on average 5 L2 cache misses and 2 TLB misses when run in L_0, whereas running the exact same code in L_1 incurs on average 400 L2 cache misses and 19 TLB misses.

5 Discussion

In nested environments we introduce a new type of workload not found in single-level virtualization: the hypervisor as a guest. Traditionally, x86 hypervisors were designed and implemented assuming they will be running directly on bare metal. When they are executed on top of another hypervisor this assumption no longer holds and the guest hypervisor behavior becomes a key factor.

With a nested L_1 hypervisor, the cost of a single L_2 exit depends on the number of exits caused by L_1 during the L_2 exit handling. A nesting-friendly L_1 hypervisor should minimize this critical chain to achieve better performance, for example by limiting the use of trap-causing instructions in the critical path.

Another alternative for reducing this critical chain is to para-virtualize the guest hypervisor, similar to OS para-virtualization [6, 50, 51]. While this approach could reduce L_0 intervention when L_1 virtualizes the L_2 environment, the work being done by L_0 to virtualize the L_1 environment will still persist. How much this technique can help depends on the workload and on the specific approach used. Taking as a concrete example the conversion of `vmreads` and `vmwrites` to non-trapping load/stores, para-virtualization could reduce the overhead for `kernbench` from 14.5% to 10.3%.

5.1 Architectural Overhead

Part of the overhead introduced with nested virtualization is due to the architectural design choices of x86 hardware virtualization extensions.

Virtualization API: Two performance sensitive areas in x86 virtualization are memory management and I/O virtualization. With multi-dimensional paging we compressed three MMU translation tables onto the two available in hardware; multi-level device assignment does the same for IOMMU translation tables. Architectural support for multiple levels of MMU and DMA translation tables—as many tables as there are levels of nested hypervisors—will immediately improve MMU and I/O virtualization.

Architectural support for delivering interrupts directly from the hardware to the L_2 guest will remove L_0 intervention on interrupt delivery and completion, intervention which, as we explained in Section 4.1.1, hurts nested performance. Such architectural support will also help single-level I/O virtualization performance [33].

VMX features such as MSR bitmaps, I/O bitmaps, and CR masks/shadows [48] proved to be effective in reducing exit overhead. Any architectural feature that reduces single-level exit overhead also shortens the nested critical path. Such features, however, also add implementation complexity, since to exploit them in nested environments they must be properly emulated by L_0 hypervisors.

Removing the (Intel-specific) need to trap on every `vmread` and `vmwrite` instruction will give an immediate performance boost, as we showed in Section 3.5.2.

Same Core Constraint: The x86 trap-and-emulate implementation dictates that the guest and hypervisor share each core, since traps are always handled on the core where they occurred. Due to this constraint, when the hypervisor handles an exit the guest is temporarily stopped on that core. In a nested environment, the L_1 guest hypervisor will also be interrupted, increasing the total interruption time of the L_2 guest. Gavrilovska, et al., presented techniques for exploiting additional cores to handle guest exits [19]. According to the authors, for a single level of virtualization, they measured 41% average improvements in call latency for null calls, cpuid and page table updates. These techniques could be adapted for nested environments in order to remove L_0 interventions and also reduce privileged instructions call latencies, decreasing the total interruption time of a nested guest.

Cache Pollution: Each time the processor switches between the guest and the host context on a single core, the effectiveness of its caches is reduced. This phenomenon is magnified in nested environments, due to the increased number of switches. As was seen in Section 4.3, even after discounting L_0 intervention, the L_1 hypervisor still took more cycles to handle an L_2 exit than it took to handle the same exit for the single-level scenario, due to cache pollution. Dedicating cores to guests could reduce cache pollution [7, 45, 46] and increase performance.

6 Conclusions and Future Work

Efficient nested x86 virtualization is feasible, despite the challenges stemming from the lack of architectural support for nested virtualization. Enabling efficient nested virtualization on the x86 platform through multi-dimensional paging and multi-level device assignment opens exciting avenues for exploration in such diverse areas as security, clouds, and architectural research.

We are continuing to investigate architectural and software-based methods to improve the performance of nested virtualization, while simultaneously exploring ways of building computer systems that have nested virtualization built-in.

Last, but not least, while the Turtles project is fairly mature, we expect that the additional public exposure stemming from its open source release will help enhance its stability and functionality. We look forward to seeing in what interesting directions the research and open source communities will take it.

Acknowledgments

The authors would like to thank Alexander Graf and Joerg Roedel, whose KVM patches for nested SVM inspired parts of this work. The authors would also like to thank Ryan Harper, Nadav Amit, and our shepherd Robert English for insightful comments and discussions.

References

[1] Phoenix Hyperspace. http://www.hyperspace.com/.

[2] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel virtualization technology for directed I/O. *Intel Technology Journal 10*, 03 (August 2006), 179–192.

[3] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Oper. Syst. Rev. 40*, 5 (December 2006), 2–13.

[4] AMD. Secure virtual machine architecture reference manual.

[5] AMIT, N., BEN-YEHUDA, M., AND YASSOUR, B.-A. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *WIOSCA '10: Sixth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*.

[6] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Symposium on Operating Systems Principles* (2003).

[7] BAUMANN, A., BARHAM, P., DAGAND, P. E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: 22nd ACM SIGOPS Symposium on Operating systems principles*, pp. 29–44.

[8] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference* (2005), p. 41.

[9] BELPAIRE, G., AND HSU, N.-T. Hardware architecture for recursive virtual machines. In *ACM '75: 1975 annual ACM conference*, pp. 14–18.

[10] BELPAIRE, G., AND HSU, N.-T. Formal properties of recursive virtual machine architectures. *SIGOPS Oper. Syst. Rev. 9*, 5 (1975), 89–96.

[11] BEN-YEHUDA, M., MASON, J., XENIDIS, J., KRIEGER, O., VAN DOORN, L., NAKAJIMA, J., MALLICK, A., AND WAHLIG, E. Utilizing IOMMUs for virtualization in Linux and Xen. In *OLS '06: The 2006 Ottawa Linux Symposium*, pp. 71–86.

[12] BERGHMANS, O. Nesting virtual machines in virtualization test frameworks. Master's thesis, University of Antwerp, May 2010.

[13] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS '08: 13th intl. conference on architectural support for programming languages and operating systems* (2008).

[14] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI '05: Second Symposium on Networked Systems Design & Implementation* (2005), pp. 273–286.

[15] DEVINE, S. W., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US #6397242, May 2002.

[16] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. In *OSDI '96: Second USENIX symposium on Operating systems design and implementation* (1996), pp. 137–151.

[17] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. Compatibility is not transparency: VMM detection myths and realities. In *HOTOS'07: 11th USENIX workshop on Hot topics in operating systems* (2007), pp. 1–6.

[18] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Network & Distributed Systems Security Symposium* (2003), pp. 191–206.

[19] Gavrilovska, A., Kumnar, S., Raj, H., Schwan, K., Gupta, V., Nathuji, R., Niranjan, R., Ranadive, A., and Saraiya, P. High-performance hypervisor architectures: Virtualization in hpc systems. In *HPCVIRT '07: 1st Workshop on System-level Virtualization for High Performance Computing*.

[20] Gebhardt, C., and Dalton, C. Lala: a late launch application. In *STC '09: 2009 ACM workshop on Scalable trusted computing* (2009), pp. 1–8.

[21] Goldberg, R. P. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems* (New York, NY, USA, 1973), ACM, pp. 74–112.

[22] Goldberg, R. P. Survey of virtual machine research. *IEEE Computer Magazine* (June 1974), 34–45.

[23] Graf, A., and Roedel, J. Nesting the virtualized world. Linux Plumbers Conference, Sep. 2009.

[24] He, Q. Nested virtualization on xen. Xen Summit Asia 2009.

[25] Huang, J.-C., Monchiero, M., and Turner, Y. Ally: Os-transparent packet inspection using sequestered cores. In *WIOV '10: The Second Workshop on I/O Virtualization*.

[26] Huang, W., Liu, J., Koop, M., Abali, B., and Panda, D. Nomad: migrating OS-bypass networks in virtual machines. In *VEE '07: 3rd international conference on Virtual execution environments* (2007), pp. 158–168.

[27] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual*. 2009.

[28] Kadav, A., and Swift, M. M. Live migration of direct-access devices. In *First Workshop on I/O Virtualization (WIOV '08)*.

[29] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: the linux virtual machine monitor. In *Ottawa Linux Symposium* (July 2007), pp. 225–230.

[30] Lauer, H. C., and Wyeth, D. A recursive virtual machine architecture. In *Workshop on virtual computer systems* (1973), pp. 113–116.

[31] Levasseur, J., Uhlig, V., Stoess, J., and Götz, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI '04: 6th conference on Symposium on Opearting Systems Design & Implementation* (2004), p. 2.

[32] Levasseur, J., Uhlig, V., Yang, Y., Chapman, M., Chubb, P., Leslie, B., and Heiser, G. Pre-virtualization: Soft layering for virtual machines. In *ACSAC '08: 13th Asia-Pacific Computer Systems Architecture Conference*, pp. 1–9.

[33] Liu, J. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IPDPS '10: IEEE International Parallel and Distributed Processing Symposium* (2010).

[34] Navarro, J., Iyer, S., Druschel, P., and Cox, A. Practical, transparent operating system support for superpages. In *OSDI '02: 5th symposium on Operating systems design and implementation* (2002), pp. 89–104.

[35] Osisek, D. L., Jackson, K. M., and Gum, P. H. Esa/390 interpretive-execution architecture, foundation for vm/esa. *IBM Systems Journal 30*, 1 (1991).

[36] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (July 1974), 412–421.

[37] Raj, H., and Schwan, K. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing* (2007), pp. 179–188.

[38] Ram, K. K., Santos, J. R., Turner, Y., Cox, A. L., and Rixner, S. Achieving 10Gbps using safe and transparent network interface virtualization. In *VEE '09: The 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (March 2009).

[39] Riley, R., Jiang, X., and Xu, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, vol. 5230 of *Lecture Notes in Computer Science*. 2008, ch. 1, pp. 1–20.

[40] Robin, J. S., and Irvine, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th conference on USENIX Security Symposium* (2000), p. 10.

[41] Rosenblum, M. Vmware's virtual platform: A virtual machine monitor for commodity pcs. In *Hot Chips 11* (1999).

[42] Russell, R. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008), 95–103.

[43] Rutkowska, J. Subverting vista kernel for fun and profit. Blackhat, Aug. 2006.

[44] Seshadri, A., Luk, M., Qu, N., and Perrig, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: 21st ACM SIGOPS symposium on Operating systems principles* (2007), pp. 335–350.

[45] Shalev, L., Borovik, E., Satran, J., and Ben-Yehuda, M. Isostack—highly efficient network processing on dedicated cores. In *USENIX ATC '10: The 2010 USENIX Annual Technical Conference* (2010).

[46] Shalev, L., Makhervaks, V., Machulsky, Z., Biran, G., Satran, J., Ben-Yehuda, M., and Shimony, I. Loosely coupled tcp acceleration architecture. In *HOTI '06: Proceedings of the 14th IEEE Symposium on High-Performance Interconnects* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 3–8.

[47] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference* (2001).

[48] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. Intel virtualization technology. *Computer 38*, 5 (2005), 48–56.

[49] Waldspurger, C. A. Memory resource management in VMware ESX server. In *OSDI '02: 5th Symposium on Operating System Design and Implementation*.

[50] Whitaker, A., Shaw, M., and Gribble, S. D. Denali: a scalable isolation kernel. In *EW '10: 10th ACM SIGOPS European workshop* (2002), pp. 10–15.

[51] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 195–209.

[52] Willmann, P., Shafer, J., Carr, D., Menon, A., Rixner, S., Cox, A. L., and Zwaenepoel, W. Concurrent direct network access for virtual machine monitors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), pp. 306–317.

[53] Yassour, B.-A., Ben-Yehuda, M., and Wasserman, O. Direct device assignment for untrusted fully-virtualized virtual machines. Tech. rep., IBM Research Report H-0263, 2008.

[54] Zhai, E., Cummings, G. D., and Dong, Y. Live migration with pass-through device for Linux VM. In *OLS '08: The 2008 Ottawa Linux Symposium* (July 2008), pp. 261–268.