

# IBM Research Report

## The Turtles Project: Design and Implementation of Nested Virtualization

Muli Ben-Yehuda<sup>1</sup>, Michael D. Day<sup>2</sup>, Zvi Dubitzky<sup>1</sup>, Michael Factor<sup>1</sup>,  
Nadav Har'El<sup>1</sup>, Abel Gordon<sup>1</sup>, Anthony Liguori<sup>2</sup>,  
Orit Wasserman<sup>1</sup>, Ben-Ami Yassour<sup>1</sup>

<sup>1</sup>IBM Research Division  
Haifa Research Laboratory  
Mt. Carmel 31905  
Haifa, Israel

<sup>2</sup>IBM Linux Technology Center



Research Division

Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

# The Turtles Project: Design and Implementation of Nested Virtualization

Muli Ben-Yehuda<sup>1</sup> Michael D. Day<sup>2</sup> Zvi Dubitzky<sup>1</sup> Michael Factor<sup>1</sup> Nadav Har'El<sup>1</sup>  
muli@il.ibm.com mdday@us.ibm.com dubi@il.ibm.com factor@il.ibm.com nyh@il.ibm.com  
Abel Gordon<sup>1</sup> Anthony Liguori<sup>2</sup> Orit Wasserman<sup>1</sup> Ben-Ami Yassour<sup>1</sup>  
abelg@il.ibm.com aliguori@us.ibm.com oritw@il.ibm.com benami@il.ibm.com  
<sup>1</sup>IBM Research – Haifa <sup>2</sup>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—virtualization and nested virtualization will become prevalent in all operating systems. 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!”*<sup>1</sup>

## 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 [26] hypervisor. As virtualization functionality becomes prevalent in commodity operating systems, nested virtualization will also be required to run those operating system/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 also enables the live migration of hypervisors with their guest virtual machines as a single entity to a different location 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 root-kits [39], hypervisor-level rootkit protection [35,41], and hypervisor-level intrusion detection [18, 25]—for both hypervisors and operating systems.

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 this 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 (multiple hypervisors) on the single level of architectural sup-

<sup>1</sup>[http://en.wikipedia.org/wiki/Turtles\\_all\\_the\\_way\\_down](http://en.wikipedia.org/wiki/Turtles_all_the_way_down).

port available. We need to address each of the following areas: CPU (e.g., instruction-set) virtualization, memory (MMU) virtualization, and I/O virtualization.

For nested CPU virtualization, we need to account for that fact that x86 virtualization follows the “trap and emulate” model [21, 22, 32]. 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 as efficient as possible. 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 [12]. 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, 16, 28, 33, 49, 50] 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 several 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, 32]. Belpaire and Hsu extended this analysis and created a formal model [9]. Lauer and Wyeth [27] 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 [10]. In contrast to to-

day’s x86 architecture which has a single-level of architectural support for virtualization, they proposed a hardware architecture with multiple virtualization levels.

The first practical implementation of nested virtualization, making use of multiple levels of architectural support, was incorporated into the IBM z/VM hypervisor [31]. Nested virtualization was also implemented by Ford et al. in a microkernel setting [15] 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 [45]). KVM [26] 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 many 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.

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 [39] 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.

## 3 Turtles: Design and Implementation

The Turtles project implements nested virtualization for Intel’s virtualization technology based on the KVM [26] 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 hardware 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 [31].

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 to decide what to do with the trap. Therefore, any trap occurring at any level from  $1 \dots n$  causes execution to drop to level 0. This limited model is implemented by both Intel and AMD in their respective x86 virtualization extensions, VMX [45] and SVM [4], and is assumed in this work.

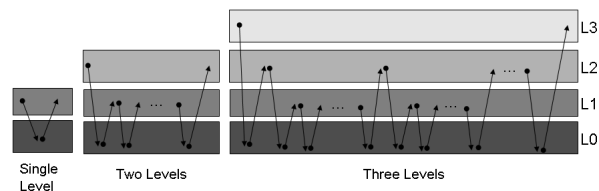


Figure 1: Nested traps with single-level architectural support for virtualization

For the sake of clarity in the exposition that follows, we limit the discussion to levels 0, 1, and 2.  $L_0$  is the bare-metal hypervisor,  $L_1$  is the guest hypervisor, which is running in a virtual machine virtualized by  $L_0$ , and  $L_2$  is a guest operating system running in a virtual machine virtualized by  $L_1$ . The principles and implementation apply equally well to  $n$  levels, with the level  $n + 1$  hypervisor or operating system ( $L_{n+1}$ ) running in a virtual machine virtualized by the level  $n$  hypervisor ( $L_n$ ).

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.

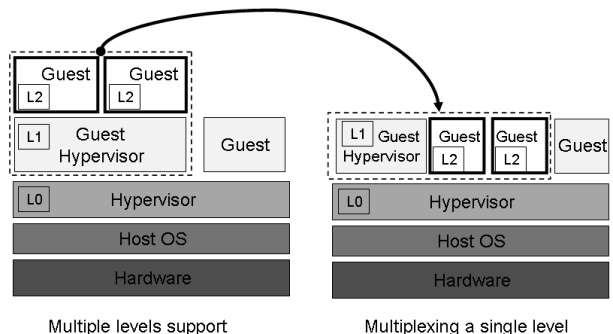


Figure 2: Multiplexing multiple levels of virtualization on a single hardware-provided level of virtualization support

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 tables. 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. See Figure 3 for the specific example of nested VMX.

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.

Likewise, when there are  $n$  levels of nesting guests, but the hardware supports fewer levels of MMU or DMA translation tables, the  $n$  levels need to be multiplexed 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 [3, 36, 37, 46]. 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 [29].

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 structures (VMCS and VMCB, respectively) which contain environment specifications for virtual machines and the hypervisor.

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  $VMCS_{0 \rightarrow 1}$  environment specification. Respectively,  $VMCS_{0 \rightarrow 2}$  is used to run  $L_2$ . Both of these environment specifications are maintained by  $L_0$ . In addition,  $L_1$  creates  $VMCS_{1 \rightarrow 2}$  within its own virtualized environment. Although  $VMCS_{1 \rightarrow 2}$  is never loaded into the processor,  $L_0$  uses it to emulate a VMX enabled CPU for  $L_1$ .

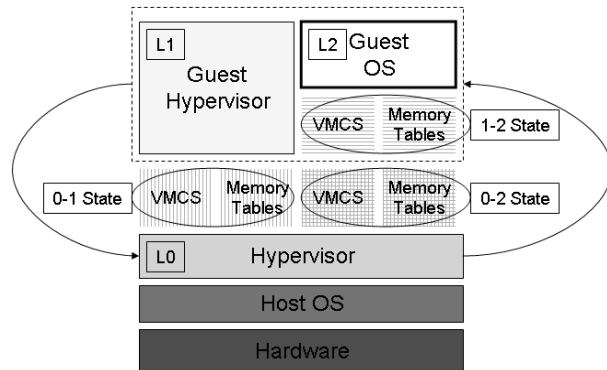


Figure 3: Extending VMX for nested virtualization

### 3.2.1 VMX Trap and Emulate

VMX instructions were designed to execute in root mode. In the nested case,  $L_1$  uses VMX instructions in guest mode to load and launch  $L_2$  guests. Any attempt to execute a VMX instruction in guest mode traps and causes a VMExit. This enables  $L_0$ , running in root mode, to trap and emulate 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 ( $VMCS_{0 \rightarrow 1}$ ) 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 structure. Because  $L_1$  believes it is running in root mode, this VMCS ( $VMCS_{1 \rightarrow 2}$ ) contains the environment of  $L_2$  from  $L_1$ 's perspective. For example, the  $VMCS_{1 \rightarrow 2}$  GUEST-CR3 field points to the page tables that  $L_1$  prepared for  $L_2$ . Clearly,  $L_0$  cannot use  $VMCS_{1 \rightarrow 2}$  to execute  $L_2$  directly, since  $VMCS_{1 \rightarrow 2}$  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  $VMCS_{1 \rightarrow 2}$  to construct a new VMCS ( $VMCS_{0 \rightarrow 2}$ ) that holds  $L_2$ 's environment from  $L_0$ 's perspective.

$L_0$  must consider all the specifications defined in  $VMCS_{1 \rightarrow 2}$  and also the specifications defined in  $VMCS_{0 \rightarrow 1}$  to create  $VMCS_{0 \rightarrow 2}$ . The **host state** defined in  $VMCS_{0 \rightarrow 2}$  must contain the values required by the CPU to correctly switch back from  $L_2$  to  $L_0$ . In addition,  $VMCS_{1 \rightarrow 2}$  host state must be copied to  $VMCS_{0 \rightarrow 1}$  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  $VMCS_{1 \rightarrow 2}$  does not require any special handling in general, and can mostly be copied directly to the guest state of  $VMCS_{0 \rightarrow 2}$ .

The **control data** of  $VMCS_{1 \rightarrow 2}$  and  $VMCS_{0 \rightarrow 1}$  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  $VMCS_{1 \rightarrow 2}$  but  $L_0$  does not trap such event for  $L_1$  (i.e., a trap is not specified in  $VMCS_{0 \rightarrow 1}$ ). To forward the event  $E_A$  to  $L_1$ ,  $L_0$  needs to specify the corresponding trap in  $VMCS_{0 \rightarrow 2}$ . 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.

### 3.2.3 VMEntry and VMExit Emulation

In nested environments, switches from  $L_1$  to  $L_2$  and switches from  $L_2$  to  $L_1$  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  $VMCS_{0 \rightarrow 2}$  that was not specified in  $VMCS_{1 \rightarrow 2}$ . 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  $VMCS_{1 \rightarrow 2}$ ). In this case  $L_0$  forwards the event to  $L_1$  by copying  $VMCS_{0 \rightarrow 2}$  fields updated by the processor to  $VMCS_{1 \rightarrow 2}$  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 the first known mechanism for virtualizing the x86 MMU [14]—used when there was no architectural support for memory management virtualization—the hypervisor is responsible for translating guest virtual addresses to host physical addresses [3, 6, 13]. Guest page tables translate guest virtual addresses to guest physical addresses. Clearly, a guest physical address does not point to the real location in the machine's DRAM because the physical memory is virtualized by the hypervisor. Thus, the hypervisor must create a new set of page tables, the *shadow page tables*, which translate guest virtual addresses directly to the corresponding host physical addresses. Next, the hypervisor runs the guest using these shadow page tables instead of the guest's page tables.

Two-dimensional page tables [12] add a new 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 guest physical address. It then uses the second table, called EPT by Intel and NPT by AMD, to translate guest physical address to 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 handling EPT violations.

The hypervisor, depending on the processors capabilities, decides 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.

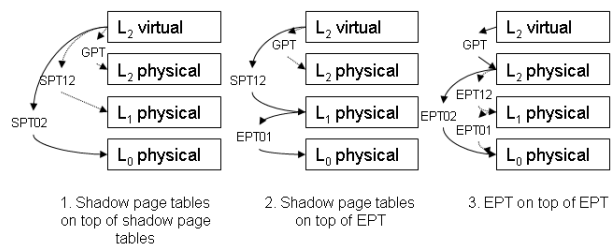


Figure 4: MMU alternatives for nested virtualization

Figure 4 shows three different nested MMU virtualization models. Shadow-on-shadow is used when the processor does not support two-dimensional page tables. Initially,  $L_0$  creates a shadow page table to run  $L_1$  ( $SPT_{0 \rightarrow 1}$ ).  $L_1$ , in turn, creates a shadow page table to run  $L_2$  ( $SPT_{1 \rightarrow 2}$ ). Obviously,  $L_0$  cannot use  $SPT_{1 \rightarrow 2}$  to run  $L_2$  because this table translates  $L_2$  guest virtual addresses to  $L_1$  host physical addresses. Therefore,  $L_0$  compresses  $SPT_{0 \rightarrow 1}$  and  $SPT_{1 \rightarrow 2}$  into a single shadow page table,  $SPT_{0 \rightarrow 2}$ . This new table translates directly from  $L_2$  guest virtual addresses to  $L_0$  host physical addresses. Specifically, for each guest virtual address in  $SPT_{1 \rightarrow 2}$ ,  $L_0$  creates an entry in  $SPT_{0 \rightarrow 2}$  with the corresponding  $L_0$  host physical address. This is possible because  $L_0$  already knows  $L_1$ 's host physical address to  $L_0$  host physical address mapping, used previously to create  $SPT_{0 \rightarrow 1}$ .

Shadow-on-EPT is used when the processor supports EPT and the  $L_1$  hypervisor uses shadow page tables.  $L_1$  uses  $SPT_{1 \rightarrow 2}$  to run  $L_2$ .  $L_0$  configures the MMU to use  $SPT_{1 \rightarrow 2}$  as the first translation table and  $EPT_{0 \rightarrow 1}$  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  $SPT_{1 \rightarrow 2}$ , and then translates from the  $L_1$  host physical address to the  $L_0$  host physical address using the  $EPT_{0 \rightarrow 1}$ . In fact, while  $L_1$  uses shadow page tables to run  $L_2$ ,  $L_0$  exploits hardware EPT support to execute  $L_2$  more efficiently.

Shadow-on-EPT has a noticeable overhead (Section 4.1.2), primarily due to page faults. Page faults cause VMExits and must be forwarded from  $L_0$  to  $L_1$  for handling. To reduce nested memory virtualization overhead to a minimum, both  $L_0$  and  $L_1$  should take advantage of EPT. This implies that  $L_0$  must expose EPT capabilities to the  $L_1$  guest hypervisor, even though the hardware only provides a single EPT table. In effect, the two EPT tables should be *compressed* into one.

Let us assume that  $L_0$  runs  $L_1$  using  $EPT_{0 \rightarrow 1}$ , and that  $L_1$  creates an additional table,  $EPT_{1 \rightarrow 2}$ , to run  $L_2$ , because  $L_0$  exposed a virtualized EPT capability to  $L_1$ . The  $L_0$  hypervisor could then compress  $EPT_{0 \rightarrow 1}$  and  $EPT_{1 \rightarrow 2}$  into a single  $EPT_{0 \rightarrow 2}$  table as shown in Figure 4. Then  $L_0$  could run  $L_2$  using  $EPT_{0 \rightarrow 2}$ , 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 substantially improving nested virtualization performance.

The  $L_0$  hypervisor launches  $L_2$  with an empty  $EPT_{0 \rightarrow 2}$  table, building the table on-the-fly. This can be done by first obtaining  $L_1$  host physical address from  $EPT_{1 \rightarrow 2}$ , then retrieving the corresponding  $L_0$  host physical address from  $EPT_{0 \rightarrow 1}$ . Finally,  $L_0$  creates an entry in  $EPT_{0 \rightarrow 2}$  that translates directly from the  $L_2$  guest physical address to the  $L_0$  host physical address.

### 3.4 I/O: Multi-level Device Assignment

I/O is one of the main challenges 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 [44], or a *para-virtual driver* is installed in the guest [6, 38], or the host assigns a real device directly to the guest which then controls the device **directly** [11, 16, 28, 33, 49, 50]. Device assignment generally provides the best performance [30, 34, 40, 50], since it minimizes the number of I/O-related world switches between the virtual machine and its hypervisor.

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

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. Implementing the best performing option, multi-level device assignment, requires handling DMA, interrupts, MMIO, and PIO for the special case of an  $L_2$  guest bypassing both hypervisors and accessing a device directly.

Handling DMA is complicated by virtualization, because guest drivers use guest physical addresses, while memory access 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. With multi-level device assignment,  $L_0$  emulates an IOMMU [5] for  $L_1$ , causing  $L_1$  to believe that it is running on a machine with an IOMMU.  $L_0$  intercepts the mappings  $L_1$  creates from  $L_2$  to  $L_1$  addresses, remaps those 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 interrupt acknowledgment will also cause guest exits. 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) are supported on the nested guest in the same way they are supported on a single-level guest [50], and can work directly in the guest without incurring exits on the critical I/O path.

### 3.5 Micro Optimizations

The main source of nested virtualization overhead is the increased number of VMExits and VMEntries. Since we assume both  $L_1$  and  $L_2$  are unmodified, we focus on improving the  $L_0$  hypervisor only, and treat  $L_1$  and  $L_2$  as black boxes. We do note however that some of these optimizations do not strictly adhere to the VMX specifications, and should be used with caution.

During a VMExit  $L_0$  spends most of the time merging the VMCS structures, as demonstrated in Section 4.3. To improve the shadowing mechanism described in Section 3.2.2, we 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 also be optimized by copying several VMCS fields at once rather than copying field by field. However, according to Intel’s specifications, reads or writes to the VMCS area must be performed using `vmread` and `vmwrite` instructions, which are field-based. Again, we empirically noted that under certain conditions one could access VMCS data directly without ill side-effects, bypassing `vmread` and `vmwrite` and copying several fields at once.

The processor keeps encoded copies of VMCS’s in memory and on-chip. By keeping shadow copies of VMCS’s in memory in exactly the same encoding used by the processor,  $L_0$  can directly transfer data between  $VMCS_{0 \rightarrow 1}$ ,  $VMCS_{1 \rightarrow 2}$ , and  $VMCS_{0 \rightarrow 2}$ . These optimizations use large memory copies as much as possible and modify individual fields directly, omitting the use of `vmread` and `vmwrite`. These direct memory accesses not only improved data transfers but also got rid of costly verifications performed by the hardware on every `vmread` and `vmwrite`.

### 3.5.1 Trapping `vmread` and `vmwrite`

Depending on the hardware implementation, dedicated instructions or regular load/store memory accesses are used to read and modify the guest and host specifications. Intel uses dedicated instructions, `vmread` and `vmwrite`, which must be trapped and emulated by the  $L_0$  hypervisor.

In contrast, AMD adopted the second model. The clear advantage of this 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. As can be seen in Section 4.3 dedicated instructions extend the nested critical path, mainly because  $L_1$  executes multiple `vmread` and `vmwrite` while it handles a single  $L_2$  exit.

One way to reduce the VMEntries and VMExits overhead is to avoid trapping on every `vmread` and `vmwrite`, for example by binary translation [3] of problematic `vmread` and `vmwrite` instructions in the  $L_1$  instruction stream. To evaluate the potential benefit of this approach, we modified  $L_1$  to use non-trapping load/stores when accessing  $VMCS_{1 \rightarrow 2}$ . The results of this optimization are described in 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 only causes guest exits. We use this synthetic, worst-case benchmark to understand and analyze the overhead and handling flow of a single  $L_2$  exit.

Our setup consisted of an IBM x3650 machine configured with a single core Intel Xeon 2.9GHz 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 compilation-type benchmark that compiles the Linux kernel in various settings. It is a general purpose benchmark. The compilation process is, by nature, CPU- and memory-intensive, and it also generates some 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 it is mostly CPU-intensive in nature.

We executed `kernbench` and `SPECjbb` in four setups: host, single-level guest, nested guest, and nested guest with non-trapping `vmread` and `vmwrite` (“Nested PV”). We used KVM as both  $L_0$  and  $L_1$  hypervisor with multi-dimensional paging. The results are depicted in Table 2.

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 discounted the Intel-specific `vmread` and `vmwrite` overhead in  $L_1$ , the overhead was 10.3% and 6.3% respectively.

To analyze the sources of overhead, we examine the time distribution between the different levels. In Figure 5 we depict 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 handle the same workload. The times are indeed similar, although

Kernbench				
	Host	Guest	Nested	Nested PV
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 PV
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

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 the various types of exits. The main reason for the cost of an exit comes 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.

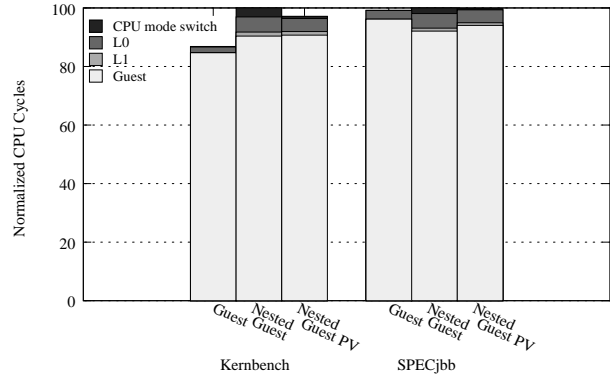


Figure 5: CPU cycle distribution

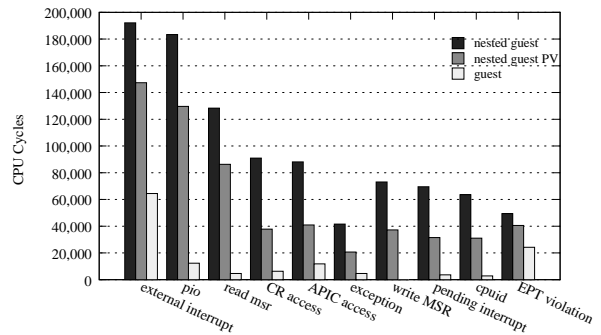


Figure 6: Cycle costs of handling various types of exits

#### 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`.

Figure 7 shows the results for running `netperf` 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 [38] for a paravirtual NIC, and a 1 Gb/s Broadcom NetXtreme II with device assignment. A single CPU core was used in all tests.

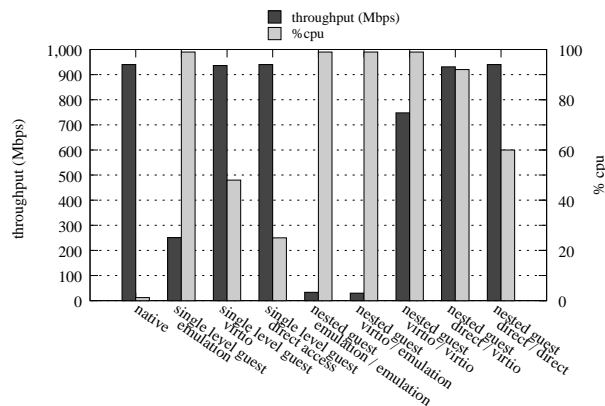


Figure 7: Performance of `netperf` in various setups

On bare-metal, `netperf` easily achieved line rate (940 Mb/s) with very little CPU being used.

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 is dominated by the cost of device emulation between  $L_1$  and  $L_2$ . The reason is that the emulation mode incurs many guest exits. In the case of the  $L_2$  guest, each exit is trapped by  $L_0$  and is forwarded to  $L_1$ .  $L_1$  executes multiple privileged instructions to handle a single  $L_2$  exit, incurring multiple exits back to  $L_0$  for each  $L_2$  exit. In this way the overhead for each  $L_2$  exit is multiplied.

Virtio 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.

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*).

Finally, we measured assigning the device all the way to  $L_2$  with multi-level device assignment. Multi-level device assignment achieved the best performance, with line rate at 60% CPU utilization (Figure 7, *direct/direct*).

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 a few percent on bare-metal. Unfortunately on current x86 architecture, interrupts cannot be assigned to guests, so both the interrupt itself and its acknowledgment 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 very close to that on  $L_0$ .

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

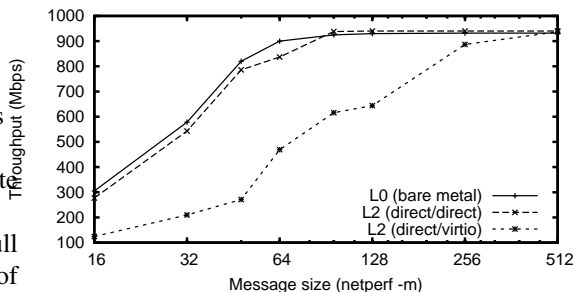


Figure 8: Performance of `netperf` with interrupt-less network driver

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 very close to that of bare-metal.

A polling driver always consumes all available CPU time, so rather than compare CPU utilization (as we did above), Figure 8 compares throughput. 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.

### 4.1.2 Impact of Multi-dimensional Paging

To evaluate the impact of using multi-dimensional paging for the  $L_2$  guest, 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  $L_2$  with the case of  $L_1$  using EPT to run  $L_2$  with multi-dimensional paging, as explained in Section 3.3.

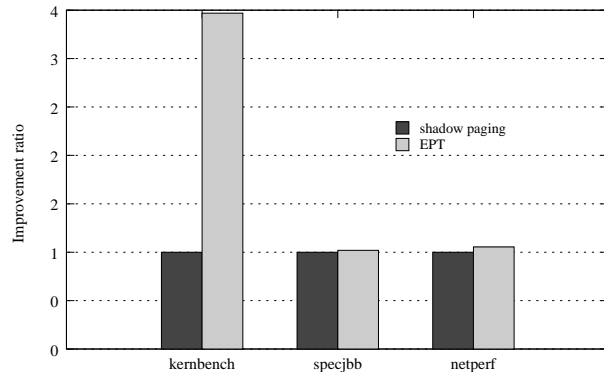


Figure 9: Impact of multi-dimensional paging

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 mostly 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 rather than KVM 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. Due to similar results obtained for VMware and KVM as the nested hypervisor, we show only `kernbench` and `SPECjbb` results. In addition, we intentionally did not install VMware tools for the  $L_2$  guest. Thus, the benchmarks were executed without any para-virtualization between  $L_1$  and  $L_2$ , increasing nested virtualization overhead.

Examining  $L_1$  exits, we noticed VMware Server uses VMX initialization instructions (`vmon`, `vmoff`,

Benchmark	% overhead vs. single-level guest
<code>kernbench</code>	14.98
<code>SPECjbb</code>	8.85

Table 3: VMware Server as a guest hypervisor

`vmptird`, `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 not performed very often. The VMware monitor module typically handles several  $L_2$  exits before switching to the Linux kernel. As a result, this behavior only slightly affected the nested virtualization performance.

## 4.3 Micro Benchmark Analysis

As a first step in understanding the performance of nested virtualization, we start with a cost analysis of handling a single  $L_2$  guest exit. For this purpose we ran a micro benchmark in  $L_2$  that only generates exits by calling `cpuid` iteratively. 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  $4 \cdot 10^6$  times and calculated the average number of cycles per iteration. We repeated the test for various 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*). Most of the cycle cost when executed in a virtual machine is due to the CPU switching from root mode to guest mode, which takes approximately 1000 cycles. When run in a nested

virtual machine it takes about 58000 cycles (Figure 10, column 2).

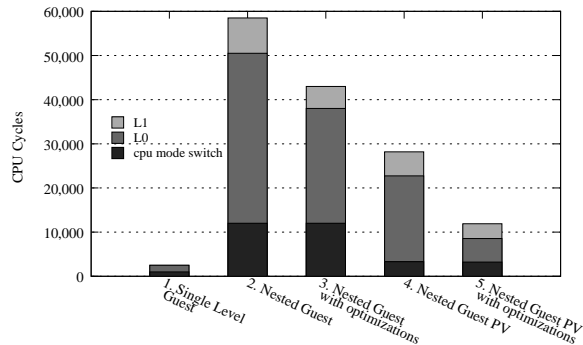


Figure 10: CPU cycle distribution for `cpuid`

It is clear that handling an exit of a nested guest is very expensive compared to handling a single-level guest exit. To understand the reason for this high cost, we analyzed the flow of handling a `cpuid` instruction by  $L_2$ . Schematically the handling can be divided into the following steps:

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  $VMCS_{1 \rightarrow 2}$   
repeat n times:
  - (a)  $L_1$  accesses  $VMCS_{1 \rightarrow 2}$
  - (b) CPU traps and switches to root mode  $L_0$
  - (c)  $L_0$  emulates  $VMCS_{1 \rightarrow 2}$  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 several 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  $VMCS_{1 \rightarrow 2}$ , as described in Section 3.2.2. In the case of

`cpuid`, in step 5,  $L_1$  reads 7 fields of  $VMCS_{1 \rightarrow 2}$ , and writes 4 fields to  $VMCS_{1 \rightarrow 2}$ , 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 1000. The total CPU switching cost is therefore around 13000 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 can reduce the virtualization overhead by 25% and by 80% 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  $VMCS_{1 \rightarrow 2}$  accesses and the corresponding VMCS access emulation, step 5. This optimization reduced the switching cost by 84.6%, from 13000 to 2000.

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 the optimized case, it takes around 5000 cycles to handle `cpuid` in  $L_1$  and around 1500 cycles to handle the same exit in  $L_0$ . Note that this 5000 cycle count does not include any exits. Moreover, the code running in  $L_1$  and in  $L_0$  is identical. The difference in cycle count is due to cache pollution and TLB flushes. Running the `cpuid` handling code incurs on average 5  $L_2$  cache misses and 2 TLB misses when run in  $L_0$ , whereas running the exact same code in  $L_1$  incurs on average 400  $L_2$  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 bet-

ter 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, 47, 48]. 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 improve depends on the workload and on the specific para-virtualization changes used. Taking as a concrete example the conversion of `vmreads` and `vmwrites` to non-trapping load/stores, para-virtualization reduced 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:** While single-level x86 virtualization extensions proved sufficient for implementing nested virtualization, there are some API implementation issues affecting the performance.

The two most performance sensitive areas are memory management and I/O virtualization. As we show in Section 4.1.2, multi-level paging increases performance; having architectural support for multi-level paging will increase performance further. Similarly, as we show in Section 4.1.1, multi-level device assignment is crucial for I/O performance; having architectural support for multi-level DMA translation tables will increase I/O performance further. Additionally, adding architectural support to deliver interrupts directly from the hardware to the  $L_2$  guest will remove  $L_0$  intervention for forwarding the interrupts and increase performance. Such architectural support will also help single-level I/O virtualization performance [30].

Some VMX features such as MSR bitmaps, I/O bitmaps, and CR masks/shadows [45] proved to be very effective in reducing exit overhead. Any architectural feature that reduces single-level exit overhead also shortens the nested critical path. These features, however, also add implementation complexity. To exploit them in nested environments, they must be properly emulated by  $L_0$  hypervisors and exposed to the  $L_1$  guest hypervisors.

As demonstrated in Section 4.3, removing the (Intel-specific) need to trap on every `vmread` and `vmwrite` instruction will give an immediate performance boost.

**Same Core Constraint:** x86 hardware virtualization extensions require the hypervisor to change guest state using the same core that the guest was running on. 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 inter-

rupted, increasing the total interruption time of the  $L_2$  guest. Gavrilovska, et al., presented techniques for exploiting additional cores to handle guest exits or to inject virtual events for a single level of virtualization [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, 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 completely 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. This performance degradation was mainly caused by cache misses due to cache pollution. Using para-virtualization guest exits could be handled by a different core, reducing cache pollution [7, 42, 43] and increasing performance.

## 6 Conclusions and Future Work

Efficient nested x86 virtualization is feasible, despite the many challenges stemming from the lack of architectural support for nested virtualization. By enabling efficient nested virtualization on the x86 platform, many exciting avenues for exploration are opened, 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 support inspired parts of this work. The authors would also like to thank Ryan Harper and Nadav Amit, for insightful comments and discussions.

## References

- [1] Phoenix hyperspace. Phoenix Technologies Ltd, <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: nineteenth ACM symposium on operating systems principles* (New York, NY, USA, 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: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), ACM, pp. 29–44.
- [8] BELLARD, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, p. 41.
- [9] BELPAIRE, G., AND HSU, N.-T. Formal properties of recursive virtual machine architectures. *SIGOPS Oper. Syst. Rev.* 9, 5 (1975), 89–96.
- [10] BELPAIRE, G., AND HSU, N.-T. Hardware architecture for recursive virtual machines. In *ACM '75: Proceedings of the 1975 annual conference* (New York, NY, USA, 1975), ACM, pp. 14–18.
- [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* (July 2006), pp. 71–86.
- [12] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII: 13th international conference on architectural support for programming languages and operating systems* (New York, NY, USA, 2008).
- [13] 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* (Berkeley, CA, USA, 2005), USENIX Association, pp. 273–286.
- [14] 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.
- [15] FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. Microkernels meet recursive virtual machines. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation* (New York, NY, USA, 1996), ACM, pp. 137–151.
- [16] FRASER, K., STEVEN, H., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe hardware access with the Xen virtual machine monitor. In *In Proceedings of the 1st Workshop on Operating System and Architectural Support for the demand IT InfraStructure (OASIS)* (2004).
- [17] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is not transparency: VMM detection myths and realities. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.
- [18] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and 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: Proceedings of the 2009 ACM workshop on Scalable trusted computing* (New York, NY, USA, 2009), ACM, 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] 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.
- [27] LAUER, H. C., AND WYETH, D. A recursive virtual machine architecture. In *Proceedings of the workshop on virtual computer systems* (New York, NY, USA, 1973), ACM, pp. 113–116.
- [28] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation* (Berkeley, CA, USA, 2004), USENIX Association, p. 2.
- [29] LEVASSEUR, J., UHLIG, V., YANG, Y., CHAPMAN, M., CHUBB, P., LESLIE, B., AND HEISER, G. Pre-virtualization: Soft layering for virtual machines. In *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific* (2008), pp. 1–9.
- [30] 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).
- [31] 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).
- [32] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (July 1974), 412–421.

- [33] 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* (New York, NY, USA, 2007), ACM Press, pp. 179–188.
- [34] 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).
- [35] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Recent Advances in Intrusion Detection*, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds., vol. 5230 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ch. 1, pp. 1–20.
- [36] 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* (Berkeley, CA, USA, 2000), USENIX Association, p. 10.
- [37] ROSENBLUM, M. Vmware's virtual platform: A virtual machine monitor for commodity pcs. In *Hot Chips 11* (1999).
- [38] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (2008), 95–103.
- [39] RUTKOWSKA, J. Subverting vista kernel for fun and profit. Blackhat, Aug. 2006.
- [40] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, J. G., AND PRATT, I. Bridging the gap between software and hardware techniques for I/O virtualization. In *USENIX Annual Technical Conference* (June 2008), pp. 29–42.
- [41] 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: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 335–350.
- [42] 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), USENIX Association.
- [43] 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.
- [44] 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* (Berkeley, CA, USA, 2001), USENIX Association.
- [45] 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.
- [46] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *OSDI '02: 5th Symposium on Operating System Design and Implementation*.
- [47] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Denali: a scalable isolation kernel. In *EW 10: Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), ACM, pp. 10–15.
- [48] 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.
- [49] 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.
- [50] 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.