Using SMT to Accelerate Nested Virtualization

Lluís Vilanova
Technion — Israel Institute of Technology
Israel
vilanova@technion.ac.il

Nadav Amit
VMware Research, Israel
namit@vmware.com

Yoav Etsion
Technion — Israel Institute of Technology
Israel
yetsion@technion.ac.il

ABSTRACT

IaaS datacenters offer virtual machines (VMs) to their clients, who in turn sometimes deploy their own virtualized environments, thereby running a VM inside a VM. This is known as nested virtualization.

VMs are intrinsically slower than bare-metal execution, as they often trap into their hypervisor to perform tasks like operating virtual I/O devices. Each VM trap requires loading and storing dozens of registers to switch between the VM and hypervisor contexts, thereby incurring costly runtime overheads. Nested virtualization further magnifies these overheads, as every VM trap in a traditional virtualized environment triggers at least twice as many traps.

We propose to leverage the replicated thread execution resources in simultaneous multithreaded (SMT) cores to alleviate the overheads of VM traps in nested virtualization. Our proposed architecture introduces a simple mechanism to colocate different VMs and hypervisors on separate hardware threads of a core, and replaces the costly context switches of VM traps with simple thread stall and resume events. More concretely, as each thread in an SMT core has its own register set, trapping between VMs and hypervisors does not involve costly context switches, but simply requires the core to fetch instructions from a different hardware thread. Furthermore, our inter-thread communication mechanism allows a hypervisor to directly access and manipulate the registers of its subordinate VMs, given that they both share the same in-core physical register file.

A model of our architecture shows up to 2.3× and 2.6× better I/O latency and bandwidth, respectively. We also show a software-only prototype of the system using existing SMT architectures, with up to 1.3× and 1.5× better I/O latency and bandwidth, respectively, and 1.2–2.2× speedups on various real-world applications.

CCS CONCEPTS

• **Software and its engineering** → **Virtual machines**; • **Security and privacy** → *Virtualization and security*.

KEYWORDS

virtualization, nested virtualization, computer architecture

ACM Reference Format:
Lluís Vilanova, Nadav Amit, and Yoav Etsion. 2019. Using SMT to Accelerate Nested Virtualization. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322261

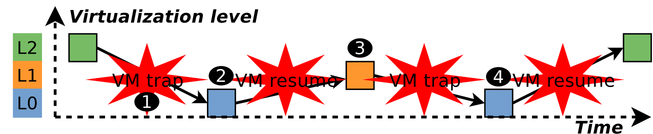

Figure 1: Nested virtualization introduces overheads when context switching between hypervisors and VMs: the host hypervisor (L0)[1] reflects a VM trap from the nested VM (L2) into the guest hypervisor (L1). Circled steps are used for reference on the text.

1 INTRODUCTION

Nested virtualization, namely running a VM inside a VM [8, 21, 22], is becoming a core technology for enterprises and data centers. This technology is already supported by leading IaaS providers, like Google and Microsoft [13, 19], to enable their clients to install nested hypervisors that provide hypervisor-enforced security [35], backwards compatibility [11, 34], easy deployment [41], hyperconvergence [45], or support for cloud native applications [10].

Nested virtualization, however, amplifies the runtime overheads incurred by existing virtualization technology, since commodity CPUs lack native support for nested virtualization. VMs are not allowed to directly execute protected operations pertaining to processor configuration state or I/O devices. As a result, hypervisors must emulate such operations through a trap-and-emulate model [40]; when a VM executes a protected operation, the processor generates a *VM trap* into the hypervisor context, which emulates the functionality of the protected instruction and then issues a *VM resume* operation, restoring the VM's context and continuing its execution. Importantly, VM trap and resume operations are costly because they require a "context switch" between hypervisor and VM states.

VM traps therefore dominate the overheads in nested virtualization, as shown in Figure 1[1]: ❶ the nested VM (L2) always traps into the host hypervisor (L0), which ❷ injects the information of the trapped event into the guest hypervisor (L1) and resumes its execution. ❸ L1 then does the actual handling of the VM trap from L2 (it is unaware of the existence of L0) and resumes the execution of L2. ❹ The VM resume operation issued by L1 triggers another

[1] Throughout this paper we use the common virtualization terminology that describes the n_{th} level of virtualization as Ln, where L0 is the host hypervisor, L1 is a guest VM/hypervisor, L2 is a nested VM, and so on. This is not to be confused with the architectural terminology that uses Ln to describe the n_{th} cache level.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA*, https://doi.org/10.1145/3307650.3322261.

VM trap into L0, which prepares the state of L2 to resume its execution. As we can see, nested virtualization multiplies the number of VM trap and resume events by **at least** a factor of 2. Oftentimes this factor is even higher since the L1 hypervisor typically triggers additional VM traps when it reads or changes the state of L2. Moreover, VM traps are even more expensive in nested virtualization as each involves saving and restoring dozens of registers, using a mixture of hardware support and complicated software dealing with the manipulation of architectural data structures.

In this paper we propose to minimize the overhead incurred by VM traps by leveraging the replicated thread execution resources present in simultaneous multi-threading (SMT) processors, eliminating the costs of single-threaded context switches between VMs and hypervisors. On the software side, we execute each virtualization level on a separate SMT thread in the same SMT core. On the hardware side, we replace the VM trap and resume operations with hardware thread stall and resume events, such that only one hardware thread is executing at any point in time. This design gives software the illusion of running on a single hardware thread, but uses the multiple hardware contexts present in SMT to accelerate context switches across virtualization levels.

The prevailing single-threaded virtualization technology must save and restore hardware contexts to and from memory, thereby incurring substantial overheads when switching between virtualization levels. In contrast, the proposed use of replicated thread states in SMT cores enables fast switching between different virtualization levels. For example, when the thread executing L2 triggers a VM trap, we stall L2 and enable the hardware context on which L0 is executing; similarly, when the thread running L0 triggers a VM resume to L1, we stall L0 and resume L1 thread's execution.

To complete our design we extend the existing SMT architecture to allow a hardware context running a hypervisor to directly manipulate the register set of a colocated hardware context that runs a subordinate VM. This feature provides a fast path for a common scenario in which the VM trap handler in a hypervisor needs to update the register context of a subordinate VM to complete the emulation of a protected instruction (e.g., increase the instruction pointer after emulating an access to an I/O device),

The proposed use of SMT cores may seem wasteful, as a core will only execute one hypervisor/VM thread at any given time. In practice, however, this is not the case since even though most enterprise and data center processors support SMT, hardware multithreading is often disabled for security and performance reasons. For one, different OSes recommend disabling SMT to thwart spectre-like and other side-channel attacks, which exploit the nature of physical resource sharing in SMT [16, 27]. In addition, process co-location with SMT often causes performance degradation [23, 33, 42] and is thus disabled for latency-sensitive workloads or when offering latency-based SLAs to third parties.

The proposed system design imposes minimal changes to existing hardware and software. Specifically, the proposed system requires fairly non-intrusive changes to the architecture, provides speedups in a way that is transparent to end-user VMs (e.g., L2), requires very modest changes on existing hypervisors (e.g., L0 and L1), and maintains the existing security guarantees of virtualization. The contributions presented in this paper are as follows:

- A qualitative and quantitative analysis of state-of-the-art nested virtualization (§ 2).
- A hardware and software co-design that accelerates context switches between VMs and their hypervisors by leveraging existing SMT resources (§§ 3 to 5). The paper also describes a software-only implementation that multiplexes hypervisors and VMs on existing SMT processors using Linux+KVM.
- A detailed evaluation that shows up to 2.6× better I/O performance on a model of our hardware/software co-design, and up to 1.55× when using our software-only prototype. With a variety of real-world applications, the software prototype can achieve speedups of 1.18–2.2× on different performance-relevant metrics (§ 6).

2 BACKGROUND AND MOTIVATION

Popek and Goldberg were the first to describe nested virtualization theoretically [17, 18, 40], and soon after processors offered hardware support for nested virtualization [7, 38]. This led to a common trade-off in computer architecture: designing complex hardware that efficiently handles most cases, or providing simpler primitives that trap into software to handle the more complex cases. As it turns out, most processors fall into the latter category nowadays.

2.1 Nested Virtualization on Single-Level Hardware Virtualization

Virtualization frameworks are usually split between a kernel-level hypervisor that manages the virtualization hardware and most performance-critical operations (like Linux's KVM [28]), and a user-level hypervisor that implements the higher-level operations like emulation of certain I/O device operations (e.g., QEMU). If we look at nested virtualization, many frameworks are based on the idea that the architecture provides a single level of virtualization in hardware, like is the case for x86 and ARM processors [2, 4, 32, 47] (which the majority of enterprises and data centers use).

In such scenarios, **every** nested VM trap goes through the host hypervisor (L0), which then either directly handles it (e.g., when the L1 guest hypervisor triggers a trap), or redirects it to one of its guest hypervisors (e.g., redirecting it to L1 when a nested VM like L2 triggers a trap). This is because whenever L1 creates a nested VM (L2), the L0 hypervisor emulates the virtualization hardware exposed to L1 using VM traps. To avoid unnecessary overheads, L0 creates a guest VM to run L2, and reflects all operations from L1 to keep L2's state up-to-date [8, 21].

Figure 2 shows a high-level overview of how the host hypervisor (L0) manages a nested VM (L2) created by a guest hypervisor (L1). For simplicity, we assume each VM has a single virtual CPU (vCPU). Hypervisors create a VM state descriptor (named *VMCS* in Intel processors) for each vCPU of its guest VMs. This descriptor contains various fields that describe information such as the reason of a VM trap (so the hypervisor can handle it accordingly), or the context of the host (L0) and its guest (L1) vCPU, which is necessary to perform context switches between L0 and L1 during VM trap and resume events; e.g., it contains the values of general-purpose, control and model-specific registers (MSRs). A VMCS does not hold the entire context of a VM, but is instead used to bootstrap a minimal amount

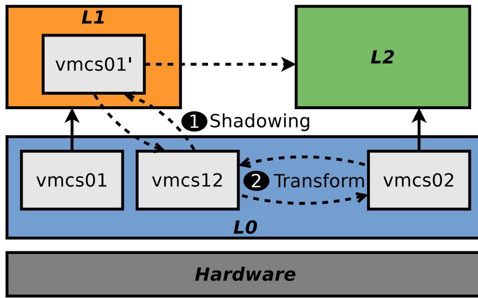

Figure 2: Relationship between the various VM state descriptors (VMCS in Intel) and the VMs they represent in nested virtualization. Numbers are used for reference on the text.

of state that allows the hypervisor to manage the rest of the context in software.

To make the example clearer, we suffix each VMCS with two numbers, the hypervisor level managing it, and the VM level it represents. For example, L0 stores the state of L1 in `vmcs01`.

When the guest hypervisor (L1) creates a nested VM (L2), it creates the corresponding VMCS for L2. Since L1 is unaware it is being virtualized (i.e., it is unaware of the existence of L0), it creates `vmcs01′` (note the tick on the name, differentiating it from `vmcs01` in L0). When L1 loads `vmcs01′` into the virtualization hardware, the processor generates a VM trap into L0. The L0 hypervisor then starts "shadowing" `vmcs01′` into `vmcs12` (step ❶), so that every update that L1 makes into `vmcs01′` traps into L0, who in turn reflects it in `vmcs12`. Note that although recent Intel processors support hardware-based "VMCS shadowing", which can be used to eliminate some common nested virtualization traps, these processors' architecture is mostly designed to run a single level of virtualization.

Finally, when the guest hypervisor L1 starts its guest VM (L2, which is a nested VM for L0), this also triggers a VM trap into L0. At this point, L0 transforms `vmcs12` into a new VM descriptor (`vmcs02`) that is used to run L2 on (❷). This is necessary because L0 *emulates* the hardware virtualization support available to L1, in order to run L2 on the real physical hardware that L0 controls. For example, a VMCS contains many pointers to physical memory addresses; therefore, the addresses set by L1 in `vmcs01′` (and shadowed into `vmcs12` by L0) contain *guest physical addresses* pertaining to L1. L0 must thus transform these addresses into the actual host physical addresses it has assigned to run L2 on. Furthermore, L1 can provide L2 with direct access to certain physical resources like the timestamp counter to track time, while the L0 hypervisor can instead choose to virtualize these resources through VM traps; this is often used, for example, to implement VM scheduling and migration. In this case, L0 configures `vmcs02` to ensure access to these resources trigger a VM trap, regardless of the configuration set by L1 in `vmcs01′`.

2.2 Life-Cycle of a Nested VM Trap

VM trap and resume operations in nested virtualization are more expensive than in single-level virtualization for three main reasons: (1) their number is amplified by **at least** a factor of two, (2) some context switches are more expensive during nested virtualization, and (3) context switches require additional VMCS transformations.

Algorithm 1 Pseudo-code to handle a nested VM trap. Right-hand side comments identify the current virtualization level and the host VMCS that they are currently using. Highlighted background shows the operations added by nested virtualization. Circled numbers reference the times in Table 1.

Line	Operation	Level — VMCS	Time
1:	...	{L2 — vmcs02}	⓿
2:	**[VM trap]**	{L0 — vmcs02}	❶
3:	vmcs12 ←transform← vmcs02		❷
4:	load vmcs01	{L0 — vmcs01}	❸
5:	inject VM trap into vmcs12		❸
6:	**[VM resume]**	{L1 — vmcs01}	❹
7:	handle VM trap using vmcs01′		❺
8:	**[VM trap]**	{L0 — vmcs01}	
9:	handle VM trap using vmcs01		
10:	**[VM resume]**	{L1 — vmcs01}	
11:	handle VM trap using vmcs01′ (cont'd)		❺
12:	**[VM resume → VM trap]**	{L0 — vmcs01}	❹
13:	load vmcs02	{L0 — vmcs02}	❸
14:	vmcs12 →transform→ vmcs02		❷
15:	**[VM resume]**	{L2 — vmcs02}	❶
16:	...		⓿

Algorithm 1 shows a complete sequence to handle a nested VM trap, with lines on a shadowed background to highlight those operations that are added during nested virtualization.

When L2 performs an operation that should be trapped by L1, it is actually trapped by L0, and accordingly performs a context-switch to L0 (Line 2). To allow L1 to handle this trap, L0 must first transform `vmcs02` into `vmcs12` to reflect any changes performed by L2 (e.g., the instruction pointer that triggered a fault is stored on the VMCS), inject the information of the VM trap event into `vmcs12` and then issue a VM resume operation on `vmcs01` to perform a context switch into L1 (Lines 3 to 6).

At this point, the CPU runs L1 code that changes `vmcs01′` and performs the necassary operations to handle the VM trap from L2 (Lines 7 and 11). As pointed above, nested virtualization induces additional VM trap and resume events when L1 performs certain privileged operations (Lines 8 to 10). Such operations include accessing control registers and certain fields of `vmcs01′`, manipulating the extended page tables [25], or reprogramming a timer interrupt. The example only shows one additional VM trap during the execution of L1's handler, but in practice this might happen **multiple** times.

After L1 finishes handling the VM trap from L2, it issues a VM resume operation on `vmcs01′`, which traps into L0 (Line 12). At this point, L0 must transform the changes performed by L1 on `vmcs12` back into `vmcs02` (remember that `vmcs12` is the shadow copy of L1's `vmcs01′`), and resume the execution of L2 (Lines 13 to 15).

Part	Time (us)	Perc. (%)
⓿ L2	0.05	0.47
❶ Switch L2↔L0	0.81	7.75
❷ Transform `vmcs02`/`vmcs12`	1.29	12.45
❸ L0 handler	4.89	47.02
❹ Switch L0↔L1	1.40	13.43
❺ L1 handler	1.96	18.87

Table 1: Time breakdown for executing a `cpuid` instruction in a nested VM (total is 10.40 us). Due to complexity, some of the context switching costs in (1) and (4) are folded into (3) and (5). Circled numbers reference Algorithm 1.

2.3 Overheads of a Nested VM Trap

To put the analysis above in perspective, Table 1 shows the breakdown of the time it takes on average to execute a `cpuid` instruction in a nested virtualization environment. We choose `cpuid` because the architecture requires it to be emulated by the hypervisor, and the handler code is short and simple. We measure the time spent in each trap handling stage in Algorithm 1 by modifying the code of the L1 and L0 hypervisors, ensuring the changes do not produce additional VM traps. We measure the time spent in L2 by measuring the execution time of the benchmark natively without virtualization. The timing measurements are then scaled to the time of executing `cpuid` in L2 without the hypervisor modifications. The experiments have a standard deviation and timing overheads below 1% of the mean with 2σ confidence, after removing outliers with 4σ confidence (evaluation platform described in § 6).

As can be seen, the L2 code (⓿), the initial VM trap, the final VM resume operation (❶) and the L1 handler itself (❺) consist of 27% of the benchmark execution time. The remaining 73% are overheads induced by nested virtualization.

These overheads are attributed to three main causes:

VMCS transformations: The VMCS transformations described in § 2.2 and Algorithm 1 (❷) account for 12.5% of the time, and are an intrinsic problem of architectures not natively supporting multiple levels of virtualization in hardware. Intel's support for VMCS shadowing provides limited benefits, as the CPU can only shadow some of VMCS fields, which do not require complicated handling (e.g., no complex address translations across VM and hypervisor spaces, or conflicting goals between a L0 and L1 hypervisor).

Context switches: Internally, every VM trap and resume event induces a pipeline flush in the processor. More importantly, the processor writes information such as CPU registers and trap information into the VMCS structure of the exiting VM, and loads registers from the VMCS into the processor in order to be able to start executing the hypervisor. This only provides a minimal execution context for the hypervisor, which must then save some of the current registers into memory, and to load others into the processor before it can start executing the actual VM trap handler. As a result, a context switch has to save and restore in excess of various dozens of values, accounting for 8% and 13.5% of the time when switching L0 between L2 and L1, respectively (❶ and ❹).

L1 exits during VM-exit handling: As explained above, the VM trap handler in L1 often triggers additional VM traps into L0. This is also the case for the `cpuid` example above, where the L1 handler triggers a VM trap when reading or writing certain fields in `vmcs01'` (the cost is folded into ❺). In practice, this example shows a best-case scenario, since L1 handlers for other types of traps trigger many more traps into L0.

Finally, note that the L0 handler (❸) takes about 47% of the time. This is because of the complexity of the code managing nested virtualization, which must emulate the virtualization of many complex architectural data structures. Importantly, some of the time pertaining to the L0 and L1 handlers (❸ and ❺) should instead be accounted into the context switching steps (❶ and ❹), increasing the cost of context switches; this is because the VM trap handlers context-switch certain VMCS fields and registers lazily.

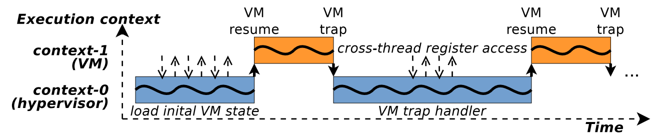

Figure 3: Executing a VM with SVT: the hypervisor executes in SMT thread 0, and starts the VM by loading its context into the SMT thread 1. Future VM trap and resume events simply switch the target SMT thread for instruction fetches, and the hypervisor can directly read and write the register state of the VM in SMT thread 1 to handle its traps.

3 DESIGN

As § 2.3 discussed, simple nested VM trap handling spends more than 20% of the time **context-switching** between the L0 hypervisor and the different VMs that it runs (see ❶ and ❹ in Table 1). During this considerable time, the hypervisor stores and loads various registers to and from memory. In practice, context-switching time might be even greater since handling most L2 traps by L1 triggers multiple traps from L1 to L0.

To eliminate the overheads of context switches, we propose *SVT* (**s**mt-based **v**ir**T**ualization), which leverages the replicated thread execution resources present in SMT processors. SVT eliminates context switches in a way that is transparent to guest VMs (i.e., L2), requires modest modifications on existing hypervisors, introduces very simple hardware changes to existing SMT processors, and does so in a way that is not riddled with the known security and performance problems that SMT entails in virtualized environments.

On the software side, the L0 hypervisor loads each virtualization level (i.e., L1 and L2) into a separate hardware context of a certain core (i.e., a core has one hardware context for each SMT thread). On the hardware side, SVT replaces the VM trap and VM resume events with hardware thread stall and resume operations, such that only one hardware thread is running at any point in time. SVT gives software the illusion that the different hardware contexts act as a single hardware thread of execution, and are therefore solely used to accelerate context switches. Figure 3 shows an example with two virtualization levels, which we will later extend to nested virtualization. Initially, the L0 host hypervisor is executing on SMT context number zero (context-0), and loads the necessary state for a L1 VM on hardware context number one (context-1). When L0

resumes the L1 VM, the processor stalls fetches from context-0, but maintains all the state on its hardware context. Since L1 already has its entire state loaded into the hardware context of context-1, the processor starts fetching instructions from it right away. Likewise, a later VM trap from L1 causes the processor to stall fetching from context-1 and to start executing instructions from context-0.

Figure 3 also shows how L0 sets up the state of L1 and manipulates it in a VM trap handler using *cross-context register accesses*. This is possible because, in existing designs, hardware threads of the same core share a single physical register file. SVT therefore only requires simple changes to allow, for example, context-0 to access the registers of context-1.

In a nested virtualization scenario, L0 would load L2's context into context-2, and L1 would be able to manipulate the registers of context-2 when handling nested VM traps. Together, these hardware modifications allow immediate context switches between virtualization levels, and avoid using memory to exchange the dozens of registers that conform the context of each hypervisor and VM.

This design poses an intermediate design point between the classical hardware designs for virtualization. On one hand, it avoids the context-switching overheads of single-level virtualization designs, found on most processors in the market. On the other hand, it avoids the complexity of full hardware support for nested virtualization, in which nested VM traps, which should be handled by a nested hypervisor (e.g., L1), do not need to be trapped by the L0 hypervisor.

3.1 The Illusion of a Single Hardware Thread with Multiple Execution Contexts

An end-user VM can *transparently* benefit from SVT when running on hypervisors that support it, and hypervisors that use SVT do not need to be aware of whether they are being virtualized themselves. This is key for adoption, since virtualization providers cannot expect their clients to change the OS of every VM they deploy.

In practice, SVT uses the multiple existing hardware contexts in SMT to accelerate nested virtualization, but provides the illusion of executing on a single hardware thread. Therefore, SVT does not need to change the logic used to schedule, create or destroy VMs; just like in existing hypervisors, state is lazily loaded into the hardware contexts every time a VM's virtual CPU (vCPU) is scheduled into a different hardware thread. Existing hardware virtualization triggers VM traps when accessing certain registers (e.g., Intel uses various VMCS fields to identify which registers will trap when accessed). SVT can apply the same technique to trap cross-context register accesses; e.g., L0 can configure certain cross-context registers to trigger a VM trap when the L1 guest hypervisor accesses them. As expected, SVT ensures that a virtualization level can only access the register context of its subordinate VMs.

Special care must be taken when dealing with interrupts. The simplest option is to have the hypervisor configure the interrupt controller in a way that treats all SVT-enabled contexts as part of the same target CPU by redirecting all external interrupts to the hardware context where the L0 hypervisor is executing.

SVT can accelerate context switches between as many nested VM and hypervisor contexts as hardware contexts are available in a core. Past that point, the hypervisor must multiplex some of the virtualization levels on a single hardware context, performing context switches between different virtualization layers.

Finally, SVT could selectively bypass some virtualization levels when triggering a VM trap to bring performance even closer to systems with full hardware support for nested virtualization, but an in-depth discussion of this topic is outside the scope of this paper.

3.2 Design Feasibility

SVT is a resource-efficient approach to accelerate virtualization because (1) SMT is available on most processors, and consumes very little real estate on the chip, (2) each SMT thread already has its own context in hardware, making context switches between SMT threads instantaneous, (3) SMT can be **easily modified** to implement our proposed changes on instruction stall and resume, and cross-context register accesses, and (4) many datacenter operators disable SMT for performance and security considerations (see § 3.4).

SVT should introduce a marginal increase of power consumption relatively to setups in which SMT was disabled, as power gating inactive thread resources in SMT has been shown to be effective [12].

3.3 Coexistence of SVT and SMT

Interestingly, one could design a system that dynamically chooses between using SMT to accelerate system-wide application execution, and SVT to accelerate VM operations on each core (SMT is known to have limited benefits on certain applications), but such analysis is out of the scope of this paper.

Finally, we present SVT as an alternative way to use the hardware resources present in SMT-enabled cores, since it springs a familiar shared mental model. It would nonetheless be also possible to implement SVT as an addition to SMT, or even as a substitute if SMT was eliminated in some processors. SVT's main structures can be reused from existing SMT designs (see § 4), and efficiently disabled in non-SVT mode.

3.4 Security and Performance Interference Considerations

Using SMT is often associated with security and performance interference problems, and therefore disabled in some enterprise and datacenter environments [23, 33, 42]. Nevertheless, the way in which SVT uses SMT is not subject to any of these problems, since an SVT-enabled core executes code from a single VM or hypervisor context at any point in time.

SMT poses a security concern due to recently found CPU vulnerabilities, such as Spectre and Meltdown, which enable attackers to leak privileged information that should not be accessible by the VM. Solutions have been developed to prevent such attacks in single-threaded CPUs with relatively reasonable performance overheads. However, some of the mitigations require software to poison (i.e., overwrite or invalidate) certain CPU micro-architectural state when it switches between different security domains, such as switches between the hypervisor and its VMs. When SMT is enabled, two different security domains might run in parallel, thereby effectively allowing one security domain to affect and monitor the other even when software poisons the CPU state. Existing solutions like Intel's STIBP induce very high overheads and are not recommended [48].

Name	Type	Purpose
SVt_visor	VMCS field	Target context for host hypervisor.
SVt_vm	VMCS field	Target context for guest VM.
SVt_nested	VMCS field	Target context for nested cross-context register accesses.
ctxtld lvl ...	Instruction	Read reg. from another context.
ctxtst lvl ...	Instruction	Write reg. to another context.
SVt_current	μ-register	Target context to fetch instructions from.
SVt_visor SVt_vm SVt_nested	μ-registers	Cached versions of the fields above.
is_vm	μ-register	Whether we are executing inside a VM. Already present in existing processors.

Table 2: Architectural and micro-architectural state changes introduced by SVt. The μ-registers are per-core.

Moreover, these solutions do not prevent all the attacks and therefore some OSes disable, or recommend disabling, SMT to thwart such attacks [16, 27].

In contrast, SMT can be safely used by SVt since the CPU would squash all speculative instructions before it starts fetching instructions of a different SMT thread. Poisoning the CPU state will therefore be effective in SVt as it is in current systems where all virtualization levels are executed on the same hardware thread. For the same reasons, SVt does not induce performance degradation due to co-executing SMT threads [23, 33, 42], given that VMs and hypervisors are never mutually contending for execution cycles.

4 HARDWARE SUPPORT

SVt reduces the latency of context switches across VMs and hypervisors by holding the state of each virtualization level (an SVt context) on a separate hardware execution context, like those provided by the SMT threads of a core (see § 3.3). SVt does so while providing the illusion of a single effective execution thread, and takes advantage of the physical shared register file commonly found in SMT core designs to allow one virtualization level to access the hardware register context of its subordinate VMs.

The overall hardware design to support SVt is shown in Figure 4. SVt adds very modest extensions to the ISA, which are summarized in Table 2: three new fields on the VM state descriptor, or VMCS in Intel, and instructions to access registers across contexts. It also performs a few simple changes on the micro-architecture: adds four per-core micro-architectural registers (three of which cache the fields above), and adds very simple logic that switches execution between hardware contexts based on VM trap and resume events.

The SVt_visor and SVt_vm fields identify the hardware contexts where a hypervisor and guest VM execute, and VM trap and resume operations switch execution to these contexts, respectively. Similarly, the SVt_nested field identifies where a nested VM of this

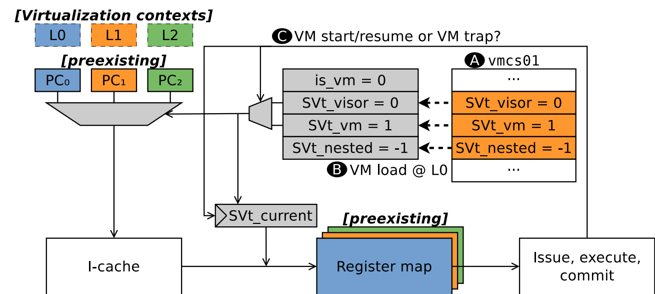

Figure 4: Overview of instruction fetch and VMCS activation in an SVt-enabled CPU with three contexts. Light gray elements control instruction fetch, while colored ones contain added or modified resources per SMT thread / SVt context. Circled letters are used for reference on the text.

guest VM executes (only used on guest hypervisors during cross-context register accesses). All three fields are cached in per-core micro-architectural registers.

The added micro-architectural register SVt_current keeps track of the "active" hardware context used for instruction fetches and execution. Its value changes according to the SVt_trap and SVt_resume registers when there is a VM trap or resume event, respectively. The existing micro-architectural register is_vm identifies whether the processor is currently executing code inside a VM.

Finally, SVt uses the lvl argument in the ctxtld/ctxtst instructions to identify the target hardware context during cross-context register accesses. A hypervisor can access the register context of a guest VM by passing argument lvl == 1, and can access the context of a nested VM by passing argument lvl == 2. Note that the target context is specified indirectly through the lvl argument; the actual context identifier is virtualized through SVt_vm and SVt_nested. When a host hypervisor is executing (is_vm == 0), passing lvl == 1 selects the context in SVt_vm, whereas passing lvl == 2 selects the context in SVt_nested. Similarly, when a guest hypervisor is executing (is_vm == 1), passing lvl == 1 selects the context in SVt_nested. Any other combination of values produces a trap into the hypervisor, which can then emulate deeper virtualization hierarchies.

To demonstrate how SVt operates, we will use an example where the L0 host hypervisor executes in context-0, and decides to execute the L1 guest hypervisor and the L2 nested VM — each with a single virtual CPU — in context-1 and context-2, respectively.

Configuring L1 and Cross-Context Register Access. The SVt_current register starts pointing to context-0, where L0 executes, and the is_vm register is zero, since we are executing on the host hypervisor. L0 then configures L1's VMCS to execute in context-1 (step Ⓐ in Figure 4) as follows: (1) in order to control the VM trap and VM resume events in L1, L0 sets the SVt_visor and SVt_vm fields to context-0 and context-1, respectively, and (2) since L1 is not running any nested VM now, L0 sets the SVt_nested field to an invalid value.

When L0 loads the VMCS into the processor (e.g., instruction VMPTRLD in Intel processors; step Ⓑ), SVt copies the newly defined VMCS fields into the corresponding micro-architectural registers.

L0 then uses the cross-context register access instructions to load the initial state of L1 into context-1 (first operation in Figure 3). L0 passes lvl == 1 to set the register context of its guest VM, L1, which runs in context-1; the ctxtld/ctxtst instructions use SVt_vm, since the is_vm register is zero. Note that the physical register file is a per-core resource shared among the different contexts, and SVt accesses the register renaming map of the target context to index into the appropriate physical register file entry (this is also true for SMT threads). Since only one context is executing in SVt at any single point in time, there is no need for additional access ports to the register maps or physical register file.

Starting L1. After loading the context for L1, the L0 hypervisor issues a VM resume. At this point (step C) SVt copies SVt_vm into the SVt_current register, effectively stalling instructions from context-0 and starting execution from context-1. A VM resume also sets the is_vm register to one to mark that a VM is executing.

Steady State: VM Trap and VM Resume. Once we are executing in the hardware context of L1 (context-1), a VM trap copies SVt_visor into the SVt_current register, switching execution back to context-0 where the L0 hypervisor resides, and also sets the is_vm register to zero. Once L0 issues a VM resume, SVt copies SVt_vm, whose value was set to context-1 above, back into the SVt_current register, and set the is_vm register back to one.

Nested Virtualization. Assuming that L1 now decides to host its own VM (L2), from its point of view L1 executes in context-0, and its guest VM (L2) in context-1. As we will see, context indexes are virtualized too, and L0 instead executes L2 in context-2.

L1 starts by setting the SVt_visor field to context-0 (where the guest hypervisor, L1, thinks it is executing), sets the SVt_vm field to context-1 (where L1 wants to execute L2), and sets the SVt_nested field to an invalid value. L1 then loads the VMCS into the processor and, like in existing processors, this triggers a VM trap into L0. At this point L0 has to perform the VMCS shadowing transformations described in § 2 between vmcs12 (vmcs01′ in L1) and vmcs02, and has to configure access to nested VMCS fields in vmcs01. L0 thus sets the SVt_nested field in vmcs01 to context-2, so that L1 can later access the registers of its guest VM in L2. Finally, L0 translates the VMCS fields in L1 (vmcs12) into their counterpart in L0 (written into vmcs02), which in this case corresponds to setting the SVt_hyper field to context-0 and the SVt_vm field to context-2 (the SVt_nested field is left with an invalid value). After this is complete, L0 resumes execution of L1.

At this point, L1 can use the instructions ctxtld/ctxtst to access the register context of L2 at context-2, even when it thinks L2 was assigned to context-1. Remember that L0 virtualized the context identifiers, and L1 accesses L2's context as identified by SVt_nested in vmcs01 (i.e., L1 passes argument lvl == 1, while the is_vm register is one).

When L1 later issues a VM resume for L2, the operation traps into L0. L0 then loads vmcs02 and issues a VM resume to it, which redirects execution to context-2 (i.e., SVt_vm == context-2).

4.1 Additional Considerations

The examples above only describe synchronous VM trap operations (e.g., when a guest hypervisor executes a VM load or resume). In a real system, VM traps can also be triggered by asynchronous events like external interrupts, which SVt treats in the same way as the examples above: an asynchronous VM trap stops the current context and switches execution to the context in SVt_visor.

Furthermore, switching across execution contexts in SVt does not require additional cache traffic, given that all contexts in a core physically share the level one data cache.

For simplicity, this hardware design has per-core resources, disallowing the execution of multiple independent VMs on the same core when SVt has been enabled on it. Nevertheless, it is quite simple to extend this design to per-context resources, such that each can independently act as a SMT thread or an SVt context, even allowing different SVt contexts of the same core to be used for different independent VMs.

5 SOFTWARE IMPLEMENTATION

To determine the software changes necessary to take advantage of an SVt-enabled processor, we will first describe the changes necessary on the hypervisor. Finally, we will describe a software-only prototype that takes partial advantage of the core concepts in SVt while using existing processors. In both cases, we will base our description on an Intel architecture using Linux's KVM module for version 4.18 (Linux's kernel-space hypervisor) and QEMU version 3.0.0 (the user-space counterpart of the hypervisor).

5.1 The SVt Hypervisor

The changes required for a hypervisor to take advantage of SVt are very limited. First, adding new VMCS fields only requires a few lines of code, mainly dealing with their index encoding, and the current VMCS layout allows fitting our three SVt_visor, SVt_vm and SVt_nested fields.

The loading of the micro-architectural registers in SVt that cache these VMCS fields already happens during the existing VMPTRLD instruction (see § 4). When a VM executes a VMPTRLD instruction, an SVt-enabled hypervisor must virtualize the context identifiers on the new VMCS fields. For simplicity, the hypervisor keeps track of the virtualization level of each VMCS, and assigns hardware context n to the nth virtualization level.

The hypervisor must also be modified to use the extended cross-context register access instructions when dealing with the register state of a subordinate VM. Fortunately, most of the relevant registers in KVM are abstracted away on a few accessor functions, which can be easily modified to use SVt.

One important deviation of SVt from the baseline KVM implementation resides in the assembly thunk that deals with VM resume and VM trap operations. This piece of assembly code contains various register saves and restores, which are not necessary in SVt; a hypervisor in SVt will simply use the cross-thread register access instructions whenever the value of a register is needed.

Finally, it is worth mentioning that KVM has various places where registers and VMCS values are manually cached on a need-to basis in order to avoid potential VM traps when accessing them multiple times. While SVt could accelerate these cases by making VM traps on accesses less expensive, the existing software caching logic does not harm correctness.

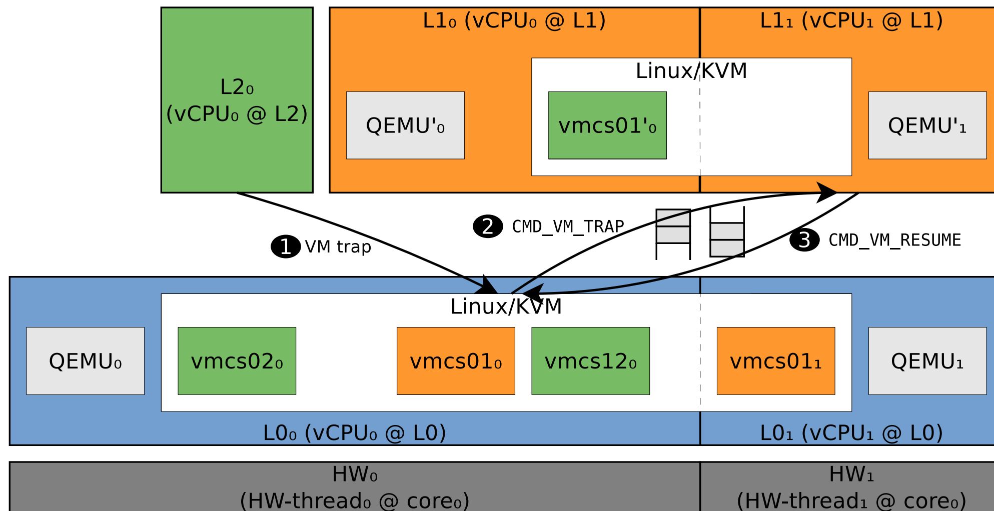

Figure 5: Operation of the software-only prototype (SW SVT) during VM trap reflection for a virtual CPU (vCPU) in L2. Subscripts denote the vCPU or hardware thread where each entity executes. Circled numbers are used for reference on the text.

Codebase	LOCs added	LOCs removed
QEMU	654	10
Linux / KVM	2432	51
Linux / other	227	2

Table 3: Summary of code changes (including comments).

5.2 Software-Only Prototype

Performing a cycle-accurate simulation of multiple nested full systems with large applications would be too computationally intensive. Therefore, we have prototyped an SVT hypervisor that runs on existing SMT processors, and Table 3 summarizes the necessary changes we performed on the KVM and QEMU codebases.

Instead of using the hardware features provided by SVT, we emulate them using software. This software prototype (SW SVT) provides some of the acceleration of SVT in a way that maintains *transparency* for nested VMs (i.e., L2), and therefore can only accelerate a subset of the cases that SVT can handle. Specifically, SW SVT only accelerates the context switches between L0 and L1 by running L1 on a separate SMT thread. Therefore, L2 runs on the same hardware thread as L0, using the pre-existing VM trap code path, and L0 and L1 communicate the information necessary to handle the VM traps of L2 using a shared memory channel.

The design of SW SVT is summarized in Figure 5, where each virtual CPU (vCPU) is associated to a VMCS structure. SW SVT pins each vCPU on its own physical hardware context, which is identified by a subscript on each of the elements in Figure 5.

When the L0 host hypervisor starts a new L1 guest hypervisor, it first creates two shared memory buffers for each vCPU. Each buffer is a unidirectional command ring that will be used to communicate VM trap and resume events regarding the L2 guest VM (commands `CMD_VM_TRAP` and `CMD_VM_RESUME` in Figure 5). L0 assigns these buffers to a `ivshmem` device in QEMU, which exposes them as PCI devices that can be mapped into regular memory in L1.

When the L1 guest hypervisor (running in $L1_0$) starts a vCPU for L2, it creates an SVT-thread ($L1_1$) that stays inside the kernel ready to serve VM traps from $L2_0$. L1 then "pairs" both threads using a hypercall to L0, so that the L0 hypervisor can reschedule them together into separate hardware contexts of the same core whenever an L2 vCPU is rescheduled into a new physical CPU. Now the system is ready to accelerate the VM traps from L2.

When L2 triggers a VM trap into L0 (❶), L0 must send a VM trap command to the SVT-thread of L1 using the shared memory communication channel (`CMD_VM_TRAP` in ❷). When the SVT-thread finishes handling the trap from L2, it responds with a VM resume command on the communication channel (`CMD_VM_RESUME` in ❸). Given that neither L0 nor L1 have access to SVT's cross-thread register access features, SW SVT sends the necessary information together with the commands on the shared memory channels between the hypervisor and the SVT-thread ($L0_0$ and $L1_1$, respectively). This information includes general-purpose register values and the VM trap identifier.

Some VM trap handlers on the SVT-thread might access resources or instructions under the assumption that L1 and L2 execute on the same hardware context; e.g., accessing certain control and MSR registers, or executing the `INVEPT` instruction. In these cases, the SVT-thread ($L1_1$) triggers a VM trap that is captured by $L0_1$. $L0_1$ then propagates the necessary information into $L0_0$ to maintain the hardware contexts in a consistent state.

L0 and L1 use the `monitor`/`mwait` instructions to efficiently wait for new commands. This is the closest we can get to the execution switch in SVT using existing processors, and our experiments validate this: `monitor` sets the CPU to watch for changes on the target communication channel, and `mwait` then stops fetching new instructions until that value changes (we configure `mwait` to keep the CPU at the C1 state). Using other forms of polling consumes cycles from the "active" hardware thread, degrading overall performance.

5.3 Avoiding Interrupt deadlocks in L1

SW SVT must be careful when interrupts arrive to $L1_0$ while an SVT-thread is handling a VM trap in $L0_1$ (this is not a problem with the hardware implementation of SVT). The following scenario would lead to a deadlock: (1) the $L1_0$ and $L1_1$ vCPUs run on the $L0_0$ and $L0_1$ hypervisor threads, respectively; (2) $L0_0$ sends a `CMD_VM_TRAP` to the SVT-thread in $L1_1$; (3) another kernel thread in $L1_1$ preempts the SVT-thread; (4) this kernel thread sends an IPI (inter-processor interrupt) to the $L1_0$ vCPU, and waits until it is handled (e.g., to perform a TLB shootdown); (5) since $L0_0$ is waiting for a `CMD_VM_RESUME` response from the SVT-thread in $L1_1$, it never resumes the $L1_0$ vCPU, leading to a deadlock ($L1_1$ is stuck waiting from an IPI response from $L1_0$).

To avoid this problem, $L0_0$ must check for new interrupts to the $L1_0$ vCPU while waiting for a response from the SVT-thread in $L1_1$. When an interrupt arrives at $L1_0$, $L0_0$ injects a new `SVT_BLOCKED` VM trap into $L1_0$; $L1_0$ then enables interrupt reception, allows the interrupt handler to proceed, and immediately yields control back to $L0_0$ through a VM resume operation on $L2_0$. After the VM resume, $L0_0$ goes back to waiting for the response from $L1_1$. This ensures forward progress in SW SVT, at the cost of longer-latency SVT command processing in the case described above. Note that correctness is not compromised: the `SVT_BLOCKED` handler in $L1_0$ never accesses the state of the $L2_0$ vCPU concurrently with the command processing of the SVT-thread in $L1_1$.

6 EVALUATION

Our evaluation consists of three parts: (1) an analysis of the performance improvements of VM trap acceleration with SVT using *micro-benchmarks*, (2) an analysis of the performance of main system operations using *subsystem benchmarks*, and (3) an analysis of end-to-end performance using real-world *application benchmarks*. All experiments use the machine parameters described in Table 4. Given that there are many storage technologies with very different latency and throughput parameters, we load the disk system image used to boot the L1 and L2 VMs into a `tmpfs`, making their accesses independent of storage technologies. All VMs are configured to avoid swapping their memory, experiments run in two virtual CPUs in L2, and all system processes are moved into a third virtual CPU using `cgroups` to avoid noise (system processes in L0 and L1 are also moved into CPUs not used for running the experiments).

The results for SVT are split into two categories. "SW SVT" shows an evaluation of the software-only prototype described in § 5.2 using the machine in Table 4. "HW SVT" shows an approximation of the hardware implementation of SVT. We modeled it by obtaining detailed timing measurements of each VM trap event and the cost of the communication channels in SW SVT; we then compared these numbers to the VM trap breakdown numbers in Table 1, and scaled the speedup assuming that every VM trap from L2 and L1 would not pay the cost of context switching (remember that SW SVT does not avoid context switches between L2 and L0).

6.1 Micro-Benchmarks

The micro-benchmarks consist of a loop with the operation under scrutiny, surrounded by a series of dependant register increments that simulate a variable workload. The loop is repeated until standard deviation and timing overheads are below 1% of the mean with 2σ confidence, after ignoring outliers with 4σ confidence.

Level	Description
L0	2×Intel E5-2630v3 (2.4GHz, 8 cores, 2-SMT), 2×64 GB RAM, Intel X540-AT2 (10 Gb)
L1	6 vCPUs, (1 reserved) 50 GB RAM, virtio-net-pci+vhost, virtio disk @ ramfs
L2	3 vCPUs (1 reserved), 35 GB RAM, virtio-net-pci+vhost, virtio disk @ ramfs

Table 4: Machine parameters. Reserved vCPUs never run our experiments, and L0 reserves a whole NUMA node.

First, we analyse the feasibility of the communication channel in SW SVT by measuring the latency of different mechanisms with different workload sizes (numbers not shown for brevity). We compare polling, `mwait` (cache-line monitoring) and mutexes against a simple function call. We also analyze different configurations where both threads are either on separate NUMA nodes, same NUMA node but different cores, or same core but different hardware threads (i.e., SMT). The main observations from these experiments are:

- *Polling* has the lowest latency for small workloads, but overheads increase with the workload in SMT because the waiting thread consumes execution cycles from the computing thread.
- Placing threads on different NUMA nodes has up to an order of magnitude longer response latency.
- Placing threads on separate cores of the same NUMA node has low response times, but that also means any thread colocated in those cores will suffer as well.
- *Mutex* has a large startup cost, but that is quickly offset in SMT as we increase the workload size because the waiting thread blocks in the kernel without consuming cycles of the working thread.
- *mwait* is slightly better than mutex in large workload sizes (Linux uses mwait when idle, but our experiment does not need to call the kernel scheduler), and has slightly longer delays with small workload sizes (mutex actively polls for a brief time first).

We can therefore conclude that SMT+mwait is a good compromise between low latency responses and low overheads when a colocated thread is performing computations.

Now we analyse how these configurations, applied to the waiting loop of the SVT-thread channels, impact performance by measuring the time it takes to execute a `cpuid` instruction in L2 (here, we use a workload size of zero). *Polling* offers very little acceleration, since the time between VM traps in L2 is always large enough that polling's overheads shadow its low response time. In contrast, the *mwait* implementation offers a reduction of around 2 us (or 1.23× speedup). This can be seen in Figure 6 (bar "SW SVT"), which compares the latency of a single `cpuid` instruction on different virtualization levels, with and without SVT. The hardware model of SVT ("HW SVT") shows an even larger speedup of 1.94×. The rest of the bars show the execution time of `cpuid` on native (L0), guest VM (L1), and baseline nested VM (L2) systems, respectively.

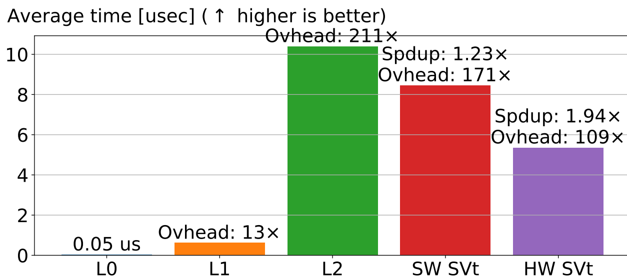

Figure 6: Execution time of a `cpuid` instruction. Speedup results for SVT are compared to the baseline L2 system.

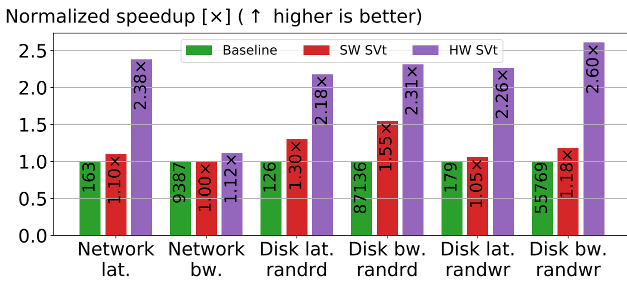

Figure 7: Speedup of SVT on various I/O subsystems.

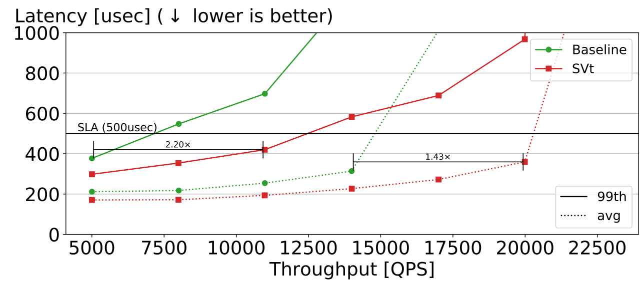

Figure 8: Latency results for *memcached* as a function of request load, using Facebook's ETC workload.

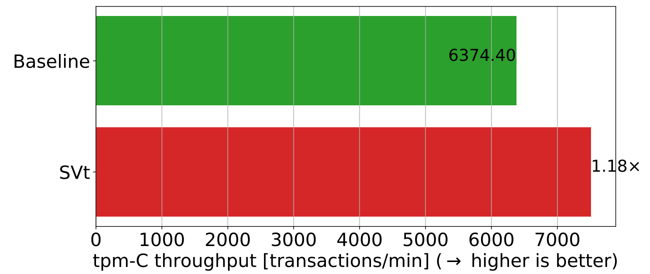

Figure 9: Throughput results for TPC-C + PotgreSQL.

6.2 Subsystem Benchmarks

Next, we measure the speedup of SVT in specific subsystems:

Network latency: TCP round-trip time (in usec) of 1 B packets using *netperf*. Measures performance of `virtio` network.

Network bandwidth: TCP throughput (in Mbps) of 16 KB packets using *netperf*. Measures performance of `virtio` network.

Disk latency: Disk latency (in usec) for 512 B accesses with either random reads or writes using *ioping*. Measures performance of `virtio` storage.

Disk bandwidth: Disk throughput (in KB/s) for 4 KB blocks with either random reads or writes using *fio*. Measures performance of `virtio` storage.

Figure 7 shows the absolute benchmark metric for the baseline L2 system and shows SVT as a speedup normalized to the baseline, using the methodology in § 6 to model the results for HW SVT.

As we can see, only the software prototype of SVT already provides speedups ranging between 1.05× and 1.55× for the device latency and bandwidth measurements (network bandwidth is close to the physical limit of 10 Gbps). Once we move to our hardware model for SVT, the speedups increase to up to a factor of 2×.

It is worth noting that some hypervisors use para-virtualization to avoid VM traps in guest hypervisors when they access a VMCS, like in the "enlightened VMCS" feature in Hyper-V [37]. SVT accelerates these cases without additional hypervisor changes, but para-virtualization can be more efficient and, therefore, used in combination. Nonetheless, profiling of our benchmarks reveals that of all time spent handling VM traps in L0, only about 4% is spent in the VM trap handlers triggered by VMCS accesses in L1.

6.3 Application Benchmarks

We now show the performance improvements of SW SVT in various real-world benchmarks, each of which stresses different subsystems.

6.3.1 Key-Value Store. The *memcached* key-value store serves us as a proxy for network latency-critical applications. We use Facebook's ETC workload [5] with the *mutilate* benchmark client [31] running on a separate physical machine with the same configuration shown in Table 4, and measure the 99th percentile and average latencies at different request loads. We consider an SLA of 500 usec, following the same parameters and constraints studied by others before [6].

Figure 8 shows the results we obtained with SW SVT. We have also profiled the baseline system to compute which VM trap handlers L0 spends most of its time in when serving L2 (numbers not shown for brevity). Depending on the target throughput, L0 spends 4.8%–19.3% of the overall time serving `EPT_MISCONFIG` traps, which largely correspond to accesses to the network device, and spends 0.5%–4.6% serving `MSR_WRITE`, largely due to configuring timer interrupts (TSC deadline MSR).

SVT acceleration results in lower and less noisy network receive and transfer latencies. This translates into a large improvement of 2.2× in 99th percentile latency within SLA, whereas average latency sees an improvement of 1.43×.

6.3.2 TPC-C Database. The *TPC-C* database benchmark serves us as a proxy for network and disk throughput. We use the *sysbench* benchmarking tool with its TPC-C addon, using a PostgreSQL database with a default configuration to serve the requests.

Figure 9 shows that SW SVT has 1.18× better transaction throughput by avoiding the costs of context switches during VM traps.

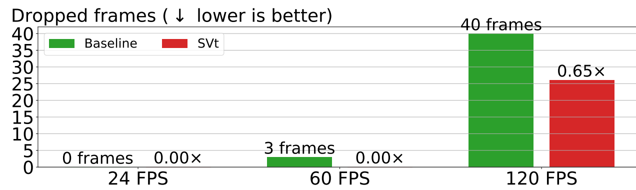

Figure 10: Video playback results (in dropped frames) as a function of video quality in frames per second.

6.3.3 Video Playback. We use a video player as a proxy for soft-realtime applications, which require timer interrupt accuracy. We use *mplayer* to reproduce the first 5 min of a 4K movie [24], which we have repackaged from 24 to 60 and 120 FPS (characteristic of HFR movies), and measure the number of dropped frames.

Figure 10 shows the number of dropped frames with all the different framerates. Profiling tells us that L0 spends 2% of the time serving `EPT_MISCONFIG` for the disk accesses in L2, and up to 1% serving `MSR_WRITE` at 120 FPS, largely due to configuring timer interrupts (TSC deadline MSR). Even if the overheads are small (L2 is idle for 61% of the time), they are enough to deliver interrupts too late for 40 frames at 120 FPS in the baseline system.

SVt brings frame drops down to 0.65× at 120 FPS, and eliminates all frame drops at 60 FPS, with a reduction of 1.12× and 1.18× for `MSR_WRITE` and `EPT_MISCONFIG` handlers, respectively.

7 RELATED WORK

During the last decade, many hardware enhancements have been proposed to alleviate virtualization overheads, mostly by reducing the number of virtualization traps. ELI proposed direct interrupt delivery to VMs without traps [20], whereas others extended this to efficiently handle more cases [46]. Self-virtualizing I/O devices were designed to eliminate traps that are triggered by VM accesses to memory-mapped I/O devices [39]. Several memory management unit (MMU) enhancements have been proposed to reduce the overheads that the additional level of memory indirection in VMs induces [1, 9, 14]. The Turtles project, which introduced nested virtualization for Intel CPUs, proposed enhancements to perform VM state reads and writes without traps in most cases [8]. Variants of most of these techniques have been adapted in commodity CPUs and production hypervisors. Since KVM is one of such hypervisors, our evaluation takes these techniques into account in both the baseline and SVt-enabled experiments.

If we look solely at I/O performance, self-virtualizing I/O devices [39] are in conflict with commonly-used live migration [50], do not easily scale with the number of VMs [26], and prevent commonly-used interposition techniques [43]. SVt triggers VM traps in a more efficient way when accessing devices, therefore supporting these use cases while reducing their overheads.

New software techniques have also been developed along the years to improve virtualization performance. Paravirtual interfaces between the hypervisor and the VMs, such as Microsoft TLFS [36] and VirtIO [44] have been employed to reduce the number of traps. These interfaces have been tailored to virtualization and are therefore more efficient than native hardware interfaces, which are based on architectural events in some cases. The overheads of *nested* virtualization have also been addressed by novel software techniques. VMware, for example, has recently improved the performance of Windows that employs virtualization based security (VBS) by 33% [49]. Despite the advances of hardware and software, nested virtualization overheads are still high and sometimes prohibitive. In addition, we evaluated SVt using VirtIO devices, therefore providing the best possible baseline.

To further improve virtualization performance, some studies proposed to offload VM trap handling onto “side-cores”. Multiple studies [3, 15, 29] have presented software techniques that can do so using existing hardware. These techniques, however, are only applicable to I/O device virtualization and increase CPU utilization and power consumption by reserving some cores to actively poll for device emulation requests. SplitX has proposed hardware enhancements for hypervisors to handle more virtualization traps asynchronously on “spare” cores in order to reduce cache pollution and obtain greater concurrency [30]. Arguably, side-core-based approaches might not be suitable in many setups, since spare idle cores are often unavailable, they are limited to VM trap events known in advance, and SplitX performance improvement through cache-pollution reduction can be offset by the overheads of inter-core communication. In comparison, SVt supports all VM trap events, including those not supported by the aforementioned software techniques (e.g., physical memory paging of a VM), accelerates VM traps without the need to reserve cores for the hypervisor, and does not depend on inter-core communication.

8 CONCLUSIONS

Nested virtualization is becoming a standard technology that eases application deployment and improves security. Despite advances along the years, the overheads of nested virtualization are still oftentimes prohibitive. Hardware support is apparently necessary to improve its performance, but the complexity of nested virtualization requires to develop basic hardware primitives that would leave most of the complexity to software.

In this paper we propose novel and simple hardware enhancements that leverage existing ubiquitous technology, SMT, to enhance nested virtualization and improve its performance. By always keeping the state of each context — the hypervisor and the VMs — cached in the CPU register files of different SMT threads of the same core, context switches are made lightweight and efficient. We propose ISA extensions that enable hypervisors to easily query and modify the state of virtual CPUs in an efficient way. We implement a software-only version of our solution on a commodity hypervisor, which enables us to validate that the proposed hardware-software interfaces are well defined. We use our prototype to evaluate the performance of our solution, and show it provides up to 2.2× speedup for real-world applications.

Indeed, the solution space for efficient nested virtualization accommodates alternative solutions. Yet, as security and performance are the key features of hardware virtualization, we believe that this solution is very compelling, as it provides a balanced trade-off between efficiency and hardware implementation complexity.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback on improving this manuscript. This research was supported by the Israel Ministry of Science, Technology, and Space and by the Israel Science Foundation (ISF grant 979/17). Lluís Vilanova was funded by Andrew and Erna Finci Viterbi and Technion Fund Post-Doctoral Fellowships.

REFERENCES

[1] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting hardware-assisted page walks for virtualized systems. In *Intl. Symp. on Computer Architecture (ISCA)*.
[2] AMD 2005. *Secure virtual machine architecture reference manual*. AMD.
[3] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conf.*
[4] ARM Ltd. 2013. *ARM architecture reference manual ARMv8-A DDI 0487A.a*. ARM Ltd.
[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS*.
[6] Adam Belay, George Prekas, , Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Symp. on Operating Systems Design and Implementation (OSDI)*.
[7] Gerald Belpaire and Nai-Ting Hsu. 1975. Hardware architecture for recursive Virtual Machines. In *ACM'75: 1975 annual ACM conference*.
[8] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, and Nadav Har'El. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Intl. Symp. on Computer Architecture (ISCA)*.
[9] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*.
[10] Google Cloud. 2017. Introducing nested virtualization for Google Compute Engine. https://cloud.google.com/blog/products/gcp/introducing-nested-virtualization-for.
[11] Oracle Cloud. 2019. Ravello. https://cloud.oracle.com/en_US/ravello.
[12] Stijn Eyerman and Lieven Eeckhout. 2014. The benefit of SMT in the multi-core era: Flexibility towards degrees of thread-level parallelism. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 591–606.
[13] Joy Fan. 2017. Nested Virtualization in Azure. https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/.
[14] Jayneel Gandhi, Mark D Hill, and Michael M Swift. 2016. Agile paging: exceeding the best of nested and shadow paging. In *Intl. Symp. on Computer Architecture (ISCA)*.
[15] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. 2007. High-performance hypervisor architectures: Virtualization in HPC systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*.
[16] Thomas Gleixner. 2019. L1 Terminal Fault. Document provided in the Linux kernel sources (`Documents/admin-guide/l1tf.rst`).
[17] Robert P. Goldberg. 1973. Architecture of virtual machines. In *Workshop on virtual computer systems*.
[18] Robert P. Goldberg. 1974. Survey of virtual machine research. *IEEE Computer Magazine* (June 1974).
[19] Google Cloud 2018. *Enabling Nested Virtualization for VM Instances*. Google Cloud.
[20] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: bare-metal performance for I/O virtualization. *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)* (2012).
[21] Alexander Graf and Joerg Roedel. 2009. Nesting the Virtualized World. Linux Plumbers Conference.
[22] Qing He. 2009. Nested Virtualization on Xen. In *Xen Summit Asia*.
[23] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten, and Martin Dixon. 2012. SMT QoS: Hardware Prototyping of Thread-level Performance Differentiation Mechanisms. In *Workshop on Hot Topics in Parallelism (HotPar)*.
[24] Ian Hubert. 2012. Tears of steel. https://mango.blender.org.
[25] Intel 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*. Intel.
[26] Intel. 2018. Introducing Intel Scalable I/O Virtualization. https://software.intel.com/en-us/blogs/2018/06/25/introducing-intel-scalable-io-virtualization.
[27] Mark Kettenis. 2018. https://www.mail-archive.com/source-changes@openbsd.org/msg99141.html.
[28] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *Ottawa Linux Symp. (OLS)*.
[29] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. 2007. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Operating Systems & Computer Architecture (WIOSCA)*.
[30] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. 2011. SplitX: Split Guest/Hypervisor Execution on Multi-Core.. In *USENIX Workshop on I/O Virtualization (WIOV)*.
[31] Jacob Leverich. 2014. Mutilate: High-Performance Memcached Load Generator. https://github.com/leverich/mutilate.
[32] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. 2017. NEVE: Nested Virtualization Extensions for ARM. In *ACM Symp. on Operating Systems Principles (SOSP)*.
[33] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Intl. Symp. on Computer Architecture (ISCA)*.
[34] Microsoft. [n. d.]. *Windows XP Mode*. https://www.microsoft.com/en-us/download/details.aspx?id=8002.
[35] Microsoft 2017. *Virtualization-based Security (VBS)*. Microsoft. https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.
[36] Microsoft. 2018. Hypervisor Specifications. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs.
[37] Microsoft 2018. *Hypervisor Top Level Functional Specification*. Microsoft.
[38] Damian L. Osisek, Kathryn M. Jackson, and Peter H. Gum. 1991. ESA/390 Interpretive-Execution Architecture, Foundation for VM/ESA. *IBM Systems Journal* (1991).
[39] PCI-SIG 2010. *Single Root I/O virtualization and sharing specification* (revision 1.1 ed.). PCI-SIG.
[40] Gerald J. Popek and Robert P. Goldberg. 1974. Formal Requirements for Virtualizable Third Generation Architectures. *Comm. ACM* (July 1974).
[41] Clear Linux Project. 2019. Clear Linux OS. Containers made simple. https://clearlinux.org/containers.
[42] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S. McKinley. 2013. Exploiting Processor Heterogeneity for Interactive Services. In *Intl. Conf. on Autonomic Computing (ICAC)*.
[43] Mendel Rosenblum and Carl Waldspurger. 2011. I/O Virtualization. *ACM queue* (Nov. 2011).
[44] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review* (2008).
[45] Scale Computing. 2018. HC3 Cloud Unity. https://www.scalecomputing.com/products/hc3-cloud-unity-cloud-platform-with-google.
[46] Cheng-Chun Tu, Michael Ferdman, Chao tung Lee, and Tzi cker Chiueh. 2015. A Comprehensive Implementation and Evaluation of Direct Interrupt Delivery. In *Intl. Conf. on Virtual execution environment (VEE)*.
[47] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. 2005. Intel Virtualization Technology. *Computer* (May 2005).
[48] Arjan van de Ven. 2018. Linux kernel mailing list discussion. https://lwn.net/ml/linux-kernel/51127fd4-5dcc-b2b9-4873-72098d2a77d9@linux.intel.com.
[49] VMware. 2018. What's new in performance? VMware vSphere 6.7. https://cloud.google.com/blog/products/gcp/introducing-nested-virtualization-for.
[50] Edwin Zhai, Gregory D. Cummings, and Yaozu Dong. 2008. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symp. (OLS)*.