Translation Pass-Through for Near-Native Paging Performance in VMs

Shai Bergman and Mark Silberstein, *Technion;* Takahiro Shinagawa, *University of Tokyo;* Peter Pietzuch and Lluís Vilanova, *Imperial College London*

https://www.usenix.org/conference/atc23/presentation/bergman

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Translation Pass-Through for Near-Native Paging Performance in VMs

Shai Bergman
Technion

Mark Silberstein
Technion

Takahiro Shinagawa
University of Tokyo

Peter Pietzuch
Imperial College London

Lluís Vilanova
Imperial College London

Abstract

Virtual machines (VMs) are used for consolidation, isolation, and provisioning in the cloud, but applications with large working sets are impacted by the overheads of memory address translation in VMs. Existing translation approaches incur non-trivial overheads: (i) nested paging has a worst-case latency that increases with page table depth; and (ii) paravirtualized and shadow paging suffer from high hypervisor intervention costs when updating guest page tables.

We describe *translation pass-through* (TPT), a new memory virtualization mechanism that achieves near-native performance. TPT enables VMs to control virtual memory translation from guest-virtual to host-physical addresses using one-dimensional page tables. At the same time, inter-VM isolation is enforced by the host by exploiting new hardware support for physical memory tagging in commodity CPUs.

We prototype TPT by modifying the KVM/QEMU hypervisor and enlightening the Linux guest. We evaluate it by emulating the memory tagging mechanism of AMD CPUs. Our conservative performance estimates show that TPT achieves native performance for real-world data center applications, with speedups of up to 2.4× and 1.4× over nested and shadow paging, respectively.

1 Introduction

Virtualization plays a central role in cloud stacks. Many academic and industry efforts strive to bring its performance closer to that of native (bare-metal) execution [19, 23, 27, 29, 37, 51, 55, 68]. Nevertheless, memory address translation in virtual machines (VMs) introduces non-trivial performance overheads. Worse, these overheads are expected to grow as applications move to larger working set sizes [26, 45], and architectures evolve to use deeper page tables to support more physical memory [1].

Memory translation in VMs (also known as guests) is performed using one of two approaches, each with its own benefits and drawbacks. In *nested paging* (see Fig. 1a), as supported by Intel EPT [51] and AMD nPT [19], VMs self-manage page tables without involving the hypervisor (also known as the host). Nested paging, however, introduces overheads during address translation: it virtualizes guest physical addresses by combining guest page tables with an additional nested page table controlled by the hypervisor. This results in up to 6× more page table entry references than a native system [19] – the MMU must issue up to 24 memory accesses to the page tables, as opposed to 4 in a native system.

In contrast, *shadow paging* (see Fig. 1b) achieves near-native translation performance. However, the guest page table management becomes costly: the hypervisor synchronizes each guest page table with a host (or shadow) page table, which directly translates guest virtual addresses (GVAs) to host physical addresses (HPAs). This avoids the translation overheads of nested paging, but introduces expensive VM exceptions to keep the page tables synchronized — often in an application's critical path.

Despite sophisticated optimizations in today's systems, such as lazy page table shadowing [61] and partial walk caches [32], we observe that workloads see up to 2.4× and 1.4× slowdowns due to nested and shadow paging, respectively (see §6). These overheads are expected to grow in future systems: applications with larger working set sizes [26, 45] will have higher TLB miss rates; emerging workloads such as function-as-a-service (FaaS) and Kata containers [34] rely heavily on process creation inside VMs, adding to page table management overheads; and upcoming CPUs will feature deeper page table hierarchies, resulting in quadratic increases in nested page table traversal overheads [1].

We explore a new approach to memory address translation, *translation pass-through (TPT)*, which enables near-native virtual memory performance in VMs. With TPT, VMs directly control translations to their assigned physical memory, without the extra level of indirection of nested paging, and without the hypervisor interventions of shadow paging during guest page table modifications. TPT is enabled by new functionality in commodity CPUs for physical memory protection using *memory tags*, e.g., in AMD SEV-SNP [60] to support confidential computing features [22]. Our key observation is that this new type of physical memory protection can be leveraged

by hypervisors to efficiently enforce memory isolation between VMs, while allowing the VMs to manage direct guest-virtual to host-physical address translation (see Fig. 1c). Thus, TPT offers a new, more efficient point in the design space across hardware-virtualized, paravirtualized, and shadow virtual memory management.

TPT's gains in memory translation performance come from the fact that one-dimensional page walks in guest VMs, combined with hardware memory protection checks, are faster than the two-dimensional page walks using nested paging. Prior work [6, 7, 16] has shown that the overheads of tag checks can be hidden by performing them in parallel with memory accesses and translation. Recent performance results on AMD SEV-SNP CPUs with physical memory tags [60] corroborate the low-performance overhead for real-world workloads. In contrast, a nested page table walk requires extra steps that are inherently sequential, making it harder to optimize.

To realize TPT, we make the following contributions:

(1) VM isolation with hardware memory protection. TPT leverages MMU support to maintain the host's physical memory frame permissions using tags. By setting per-VM frame tags, we can safely allow VMs to manage direct guest-physical-to-host-physical page tables: the hypervisor ensures that a VM can only access host frames assigned to it, regardless of the host physical addresses in the VM's page tables ("Hypervisor" and "HW" layers in Fig. 1c). Existing AMD CPUs with SEV-SNP already support the host frame permissions we need for TPT; we cannot use SEV-SNP as-is because frame tags are coupled with nested paging and expensive memory encryption, but we would require only simple hardware changes: adding two registers to configure TPT, and enabling the frame tag functionality separately from the rest of SEV-SNP.

(2) Selective user-space translation. Enabling TPT for an entire VM would require a fundamental redesign of the boot process, memory management, and I/O in the guest OS. Fortunately, TPT's performance benefits are largest for user-space applications with large working sets, but are less so for small working-set applications or kernel-space (see §3). The guest OS thus enables TPT only in user-space execution of some processes, which are dynamically identified to take advantage of it. We achieve this by introducing a new type of *TPT page table* with GVA-to-HPA translations that are checked against the VM's host frame permissions, whereas guest kernel code and other non-TPT-enabled processes use the traditional *non-TPT page table* (similar to how PTI works [58]). This supports incremental deployments in which TPT and non-TPT processes and VMs co-exist on a host and with minimal guest OS changes, which simplifies the deployment of TPT.

(3) Hypervisor-compatible extensions. We describe a design for TPT that is compatible with existing hypervisors. TPT only requires modest changes to the KVM interface: it exposes the physical memory map to enlightened guests, and

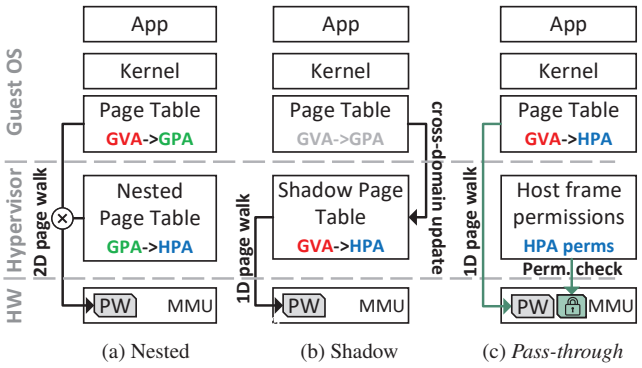

Figure 1: Existing (nested, shadow) and proposed pass-through paging approaches (GVA/GPA mean guest virtual/physical address; HVA/HPA mean host virtual/physical address; PW means page walk.)

extends the guest pvops backend in Linux to seamlessly incorporate the extra TPT page tables. Our design permits the hypervisor to retain control over guest physical memory without introducing performance penalties: the hypervisor can use existing memory ballooning techniques, and can forcibly reclaim host frames from uncooperative VMs. To support host frame reclamation and VM migration, the guest OS keeps a pair of synchronized TPT and non-TPT page tables, which we call *dual page tables*, for each TPT-enabled process; using pvops in the guest OS keeps synchronization transparent and with low overhead. Since a dual page table is always kept in sync, the hypervisor can force any guest process to utilize its non-TPT page table while a host frame reclamation or VM migration is underway.

We implement TPT using a Linux guest and KVM/QEMU hypervisor. We evaluate our TPT prototype using a commodity x86–64 CPU — which does not perform any host frame permission check, but can execute applications much larger than a traditional CPU simulator —, and assume an optimized MMU implementation that executes permission checks in parallel with page table traversal. We also model the performance of a naive MMU implementation where operations are executed in sequence by injecting additional delays in page table walks, and discuss how both approaches reasonably model the overheads that we should expect from a hardware implementation such as is contained in SEV-SNP.

Our results show that an optimized TPT implementation achieves native performance, and is 2× and 1.2× faster than nested paging and shadow paging, respectively, on a PageRank benchmark. Even with a naive MMU implementation, TPT exhibits a geometric mean slowdown of only 3% over native execution for a series of typical cloud workloads, including Memcached and kernel compile.

The TPT implementation is available as open source software at https://github.com/acsl-technion/TPT.

2 VM Address Translation

We discuss the properties of current memory virtualization approaches (§2.1) and motivate the opportunities offered by new hardware protection mechanisms in recent CPUs (§2.2).

2.1 Memory virtualization approaches

Current VMs use one of the three following mechanisms:

Shadow paging uses hypervisor-managed *shadow page tables*, shown in Fig. 1b, that directly translate a guest virtual addresses (GVA) to host physical addresses (HPA). The guest maintains its own page tables, but the hypervisor forces the MMU to use shadow page tables for address translation. Shadow paging thus offers native translation performance with a one-dimensional page walk.

The hypervisor typically write-protects guest page tables, such that *every guest write* to a guest page table traps into the hypervisor to update the shadow page table [4]. Modern implementations thus need to trap on guest page table writes and on privileged guest instructions, such as TLB flushes. Despite elaborate optimizations [61], shadow paging suffers from these high intrinsic costs for page table manipulation.

The performance of page table manipulation is critical for some workloads, such as function-as-a-service (FaaS). With FaaS, process initialization is on the critical path of function invocations, which includes page table manipulations [25]. To achieve strong isolation, FaaS runtimes are commonly deployed in VMs, e.g., Kata containers [34, 52], which makes page table management a performance-critical operation.

Paravirtualization of MMUs, e.g., in Xen-PV [14], predates hardware virtualization extensions. It can be seen as a variant of shadow paging in which traps are replaced by explicit hypercalls in the guest OS, used to request changes to the hypervisor-managed GPA-to-HPA page tables.

Paravirtualized page tables, however, are costly: hypercall overheads are of the same magnitude as the traps in shadow paging, requiring context switches between VMs and the hypervisor. While paravirtualization can batch modifications to reduce overheads, lazy shadow paging can achieve similar benefits. Therefore, only older hypervisors used paravirtualized page tables by default [12, 14], newer ones use optimized shadow paging and nested paging [13, 66, 73, 75].

Nested paging is a hardware-accelerated approach that performs GVA-to-HPA translation using two hierarchies of page tables: (i) guest (VM-controlled) page tables and (ii) host (hypervisor-controlled) page tables (see Fig. 2). The guest page tables translate GVA-to-GPA (guest physical addresses); the host ones translate GPA-to-HPA. Every GPA in a guest page table requires a GPA-to-HPA translation by the MMU. This procedure is called *two-dimensional page walks* [19].

A two-dimensional page walk multiplies the number of memory accesses per address translation. In the worst case, a single translation must access m levels of the guest page table (horizontal dimension in Fig. 2), where the GPA of each level

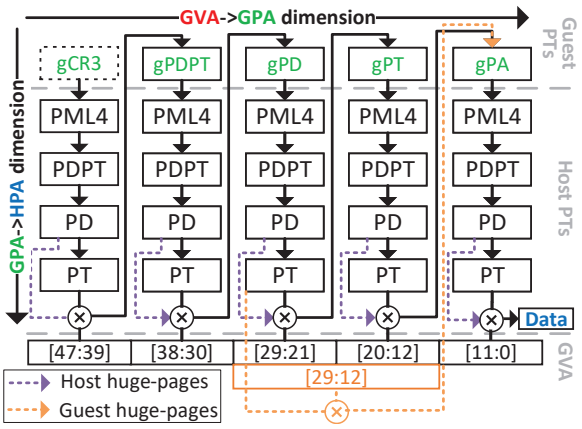

Figure 2: 2-D page table and page walk for nested paging

is first translated by accessing n levels of the host page table (vertical dimension), plus n and m accesses to the contents of the respective page tables: $nm+n+m$ memory accesses in total (e.g., 24 memory access in existing x86–64 processors using 4 KB pages, where $m=n=4$).

Several studies have reported that the overheads of two-dimensional page walks may account for over 30% of application execution time [19, 57]. With the growth in working set sizes and deeper radix page tables [1], this could lead to a quadratic increase in memory virtualization overheads.

Translation overheads can be reduced by using huge pages on the host and/or guest page tables. This bypasses part of the page walk, as shown in the dotted lines in Fig. 2. Their use, however, is not always feasible and may lead to under-utilization of memory due to internal fragmentation [54]. Hardware partial walk caches target similar optimizations but are typically less effective due to their reliance on spatial and temporal reuse [32].

Despite its higher translation costs, nested paging is often the preferred virtualization approach, because it enables guests to perform page table updates without hypervisor intervention while remaining compatible with full virtualization.

2.2 Hardware memory protection

Physical memory protection, recently introduced in commodity CPUs, offers a new hypervisor-controlled mechanism for memory isolation across VMs [60, 64]. AMD SEV-SNP [60] is one example of such technology, which utilizes both memory encryption and physical memory tagging to enhance VM isolation; other architectures, e.g., RISC-V, also offer mechanisms for physical memory protection [71].

In AMD SEV-SNP, the MMU checks each host physical memory access against a *host frame permission table* (called *RMP*) that identifies which VM can access each host frame. The RMP is a physically-contiguous array of memory that contains one entry per host frame. Each entry has a unique identifier of the VM that the host frame is assigned to. Since the MMU checks every HPA against the RMP, this ensures that VMs only access HPAs assigned to them.

RMP checks only happen during a TLB miss, and AMD's implementation has various optimizations to reduce their overhead: (1) RMP entries can be cached as regular data when accessed by the MMU during a page walk, minimizing RMP memory accesses (page table entries can be cached too); and (2) cache lines are extended with their RMP entry to eliminate RMP lookups on cached data.

To decide if it is possible to leverage such hardware memory protection features to accelerate address translation and page table manipulation in VMs, we can consider existing AMD SEV-SNP deployments. SEV-SNP is integrated into Microsoft's Azure cloud platform, and recent performance results show a low overhead for SEV-SNP-enabled VMs [48]. Since SEV-SNP performs both host frame tagging and cache line encryption, with the latter dominating performance overheads [49], using just host frame tagging as part of memory translation should have an even lower overhead (see §5.4).

3 Translation Pass-Through Design

Our design goals and key insights for TPT are as follows:

Native performance. Our solution should offer efficient translation in both current and future systems, where we expect existing memory virtualization approaches to not scale (see §2). Our insight is that, unlike the quadratic overhead of nested paging, VM translation with host physical memory tagging adds a single access to the tag for each page table level. Prior work has shown that such overheads can be largely hidden at the micro-architectural level [6, 7, 16] (see §5.4), and existing commercial results seem to indicate the same [48]. Thus, we make a choice to use tagged physical memory to achieve native translation performance in VMs.

Compatibility with hypervisors/guests. To facilitate adoption, our solution should avoid major changes to existing hypervisors and guest OSs. Achieving this is challenging, as memory translation is deeply ingrained in hypervisor and guest OS implementations. Paravirtualization is often used in virtualized environments in which full hardware virtualization is too complex to implement or too expensive [30, 50, 59]. Nevertheless, a fully paravirtualized memory management interface, such as Xen-PV [74], would require extensive changes to the guest OS, including the boot sequence, I/O layer, and kernel memory management.

To sidestep this complexity, our observation is that TPT's translation approach can be confined to user-space applications, which experience the highest gains. It can be enabled dynamically for each guest process at runtime. As we show in §5.2, TPT is not expected to benefit kernel performance. By limiting TPT use to user-space, we avoid changes to I/O management, guest system boot or guest memory management, and maintain compatibility with existing hypervisor interfaces and host memory management features, such as VM migration or host frame reclamation.

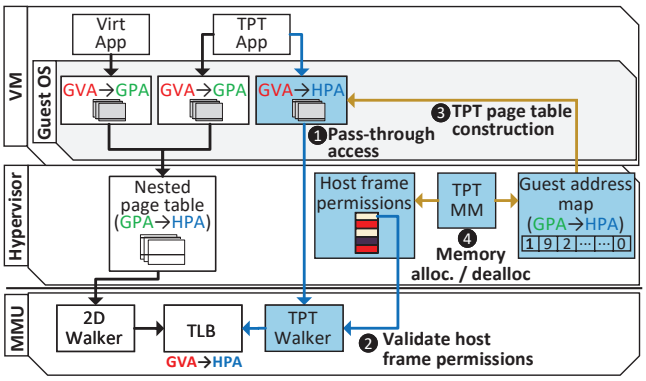

Figure 3: TPT prototype design (Extensions to hardware, hypervisor, and guest are shaded. Memory fast-path translation are blue arrows. Page table management are yellow arrows.)

3.1 Design overview

Fig. 3 shows the main components of our TPT prototype. As a starting point, we assume that the guest OS and hypervisor use nested paging by default; however, our design is general and also applicable to shadow paging (e.g., Linux supports both) or a hybrid system [27, 50]. Further, we assume the availability of hardware memory protection using host frame tags as used internally in AMD SEV-SNP (see §2).

By default, every guest process has a single non-TPT page table (as usual; see "Virt App" in Fig. 3), until the TPT prototype enables TPT on that process. At this point, the guest OS constructs and maintains *dual page tables* for that process, by keeping both a TPT and non-TPT page table in sync (see "TPT App"). We could instead have one or the other depending on whether we enabled TPT on each process, but maintaining dual page tables is inexpensive (evaluated in §6.3), keeps change complexity low, and makes host frame revocation simple to implement: e.g., during host frame reclamation or VM migration, the hypervisor can force all processes to use their non-TPT page table until the guest OS has "repaired" the corresponding TPT page tables (see § 3.2, 3.4 and 4.2).

The guest OS uses the non-TPT page table as the canonical representation of address translation. It only maintains a TPT page table for user-mode execution of processes for which it explicitly requested TPT (step ❶). When running with TPT, the guest updates both page tables, thus keeping them in sync.

To populate the TPT page table, the guest OS retrieves GPA-to-HPA translations from the "guest address map" (step ❸). The guest address map is a data structure used by a VM to know the mapping between its GPAs and the corresponding HPAs. The hypervisor maintains one guest address map for each VM, maps it as a read-only guest physical memory range when a VM boots, and updates it each time the hypervisor changes a guest-to-host physical memory assignment for that VM (step ❹). This approach minimizes changes to the hypervisor since we can reuse GPA faults and existing balloon drivers to manage host frames, and to update the corresponding guest address map and host frame permission table.

The design of dual page tables assumes that each hardware thread has registers pointing to both page tables (TPT and non-TPT), and that the hypervisor may force a VM to use its non-TPT page tables despite also having TPT page tables. We describe the necessary hardware extensions in §4.

3.2 Dual page tables in the guest OS

Every hardware thread has hardware support to access the dual page tables, by using separate page table pointer registers. This ensures compatibility: non-TPT VMs require no changes at all, and TPT-enabled VMs boot without changes — i.e., no TPT page-table is used. A TPT VM then dynamically enables TPT by providing a TPT page table, and can also disable it.

The guest OS enables TPT on a per-process basis, and for user-level code only, where TPT is most effective (see §5.2). TPT can be activated for a process based on an explicit user request. One could extend this with automated runtime policies, although this is out of the scope of this paper; e.g., by monitoring performance metrics such as TLB-miss rates, RSS values, and page-walk cycles.

The guest OS always operates on the canonical non-TPT page table, with its GVA-to-GPA translations, to avoid changes on existing components such as memory management abstractions and algorithms, e.g., to perform reverse virtual address lookups. Each time a non-TPT page table is modified, the guest OS efficiently reflects the changes to the corresponding TPT page table, if any. Entries in a TPT page table take the canonical GPA and translate it to the corresponding HPA using the added guest address map in step ❸, which provides "GPA→HPA" translations specific to this VM.

3.3 Page walks and host frame permissions

The hypervisor assigns a unique identifier to each VM, which is used in the host frame permissions table to mark which host frames are assigned to each VM. When the MMU traverses a TPT page table, it raises an exception into the hypervisor whenever the tag for the VM does not match that of an accessed host frame.

Note that a 4-level page walk in TPT incurs up to 9 memory accesses, but allows micro-architectural optimizations to hide permission checks (see §5.4). If we instead look at a 5-level page table, nested paging goes from 24 to 35 memory accesses, but TPT only goes from 9 to 11 memory accesses, highlighting the advantage of TPT when moving to upcoming architectures with larger physical memory spaces [1].

The behavior of a non-TPT page table is unchanged (see "GVA→GPA" in Fig. 3): a TLB miss triggers a traversal from the MMU, which performs a two-dimensional traversal when using a nested page table set by the hypervisor.

3.4 Host frame management in the hypervisor

The hypervisor tracks GPA→HPA assignments as usual: guest accesses to an unassigned guest frame trigger allocation and mapping into a host frame, i.e., via guest access to an unmapped page in the EPT.

These host frame assignments are captured by the hypervisor's TPT *memory manager* ("TPT-MM" in step ❹). It then updates the host frame permissions used by the MMU in step ❷ and the per-VM guest address map used by the guest OS in step ❸ (see §4.2 for more details).

The hypervisor reclaims host frames from a VM by using the existing balloon driver. When frames are released to the hypervisor, the latter updates the guest address map and host frame permissions accordingly, followed by the invalidation of the TLB entries using existing mechanisms – nested paging uses instruction `INVEPT` in x86–64, whereas shadow paging uses a reverse map to invalidate individual pages.

In some cases, the hypervisor must forcibly reclaim host frames without guest OS cooperation (e.g., the VM is uncooperative or its balloon driver is slow to respond). We design a protocol between the guest OS and hypervisor to handle this case: (1) the hypervisor forcibly disables the use of TPT page tables on that VM and injects a "TPT status" exception onto it. Since the guest has dual page tables, the processor will exclusively use the non-TPT page tables; (2) the hypervisor reclaims any host frames it needs from the VM as usual, removes them from the guest address map, and resets the host frame permissions; (3) the hypervisor resumes guest OS execution; (4) the guest OS gets the injected interrupt and "repairs" the affected page tables to ensure that they do not use the reclaimed host frames; and (5) the guest OS issues a hypercall to notify the hypervisor it can re-enable TPT.

VM migration is handled similarly. The hypervisor disables TPT during migration and notifies the guest OS upon completion by injecting the "TPT status" exception. At this point, the guest OS repairs its TPT page tables based on the new guest address map contents, and re-enables TPT.

Note that this protocol is only needed for TPT-enabled guest processes, which we expect to be a small fraction of all VMs and guest processes. It also only triggered in already expensive cases, such as forceful host frame reclamation, and VM migration. Failure to unmap reclaimed host frames is not a security issue: the VM does not have access to such frames through the host frame permissions, and guest access to an unassigned frame results in an exception in the hypervisor, which can allocate a new frame or terminate the VM.

3.5 I/O host frames and pass-through devices

The host frame permission table only covers the system's main memory address range, which prevents support for pass-through devices on TPT processes (e.g., a DPDK application with VM device pass-through [2, 24]).

To support such additional physical memory address ranges, TPT includes new privileged address range registers, which are configured by the hypervisor to grant a VM access to the selected ranges (selected during VM boot). These ranges are assigned to the executing VM, and the hypervisor exposes their HPAs through the guest address map. This mechanism operates similarly to x86 MTRRs [32] and AMD's IORRs [8].

Name	Description
TPT-cr3	Root TPT page table location (zero disables TPT)
cpuid leaf	Discovery for TPT support
tag_{base,end}	MSR w/ physical addr. of host frame permissions table
VMCS TPT-tag	Tag associated with VM (zero disables TPT)
VMCS TPT-IORR	Address range regs. for VM permissions to HPA ranges
TPT-fault	Exception for invalid permission access
TPT_enable	Hypercall to request TPT enable
TPT_status	Hypervisor-injected int. to signal TPT status change
TPT_addrmap	Virtual PCIe device with guest address map

Table 1: TPT interface added to CPU (Guest ISA (top); hypervisor ISA (middle); guest/hypervisor interface (bottom).)

4 Implementation

We implement a prototype of TPT for Linux 5.16 that consists of 1,700 lines of code (LoC) for the guest OS extensions, 500 LoC in the KVM hypervisor, and 700 LoC in QEMU to configure and start VMs (counted using CLOC [21]). Our prototype targets x86-64, but most changes are architecture-agnostic. Table 1 summarizes the changes visible at the ISA and guest/hypervisor interface, where VMCS identifies the VM hardware configuration fields. In particular, dual page tables are implemented by setting both cr3 and the new TPT-cr3 (which can be disabled by the hypervisor by setting VMCS field TPT-tag to zero).

4.1 Hypervisor extensions

Our hypervisor is based on Linux KVM/QEMU and supports shadow and nested paging by default.

Host frame permissions. The host frame permission table is located in contiguous host physical memory, spanning as many entries as frames in the host physical memory range. With 32-bit tags (already used by AMD SEV-SNP [60]), that corresponds to a 0.1% memory overhead.

The table is configured by the hypervisor, which sets registers tag_{base,end} (similar to AMD RMP_{BASE,END} [8]). In addition, the hypervisor provides a unique TPT identifier for each VM in VMCS field TPT-tag (checked against host frame permission table entries), or sets it to zero to disable TPT (e.g., when migrating a VM or forcing page reclamation).

Extra memory ranges are permissioned by the hypervisor via VMCS TPT-IORR (e.g., for user-level device passthrough; see §3.5), which are used when the requested address is outside the DRAM's physical address space.

Guest address map. The hypervisor generates a mapping for every VM to translate the VM's GPAs to their corresponding HPAs. Upon booting the VM, the hypervisor constructs the guest address and maps it as a read-only guest physical memory range in the VM. Each map is an array in the host virtual memory that covers the guest physical memory range and extra pass-through device ranges assigned to the VM (configured via QEMU). This map is exposed as a virtual PCIe device to VMs (TPT_addrmap), and the size of each map entry is 8 B per 4 KiB frame, resulting in 0.2% memory overhead.

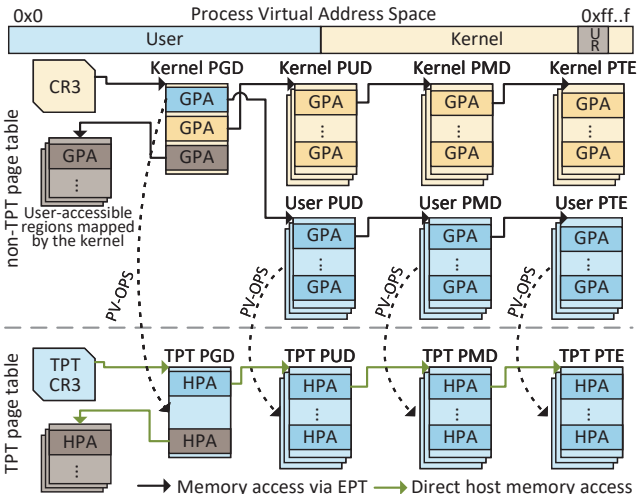

Figure 4: Dual page tables in guest OS (Changes in non-TPT page tables are synchronized to TPT page tables via pv_ops.)

We extend KVM's tdpmmu [63] to update the guest address map's contents when modifying GPA mappings without allocating hypervisor memory on unmapped guest sub-ranges.

4.2 Guest OS extensions

The guest OS changes are largely restricted to a new TPT-specific paravirtualization backend. The new features are activated when the guest OS kernel detects TPT support by the hypervisor (via a new cpuid leaf, configured by the hypervisor). After the discovery of TPT, the guest OS maps the TPT_addrmap device as a write-back (cacheable) memory range as its guest address map.[1]

To enable TPT for a process, the guest user writes to a new procfs entry, which triggers the construction of dual page tables. After the TPT page table has been created, the kernel puts the TPT page table into the new TPT-cr3 register to activate it when a process is rescheduled.

Dual page tables. Our guest OS maintains dual page tables, shown in Fig. 4, and the TPT page tables only cover addresses accessible in user-mode. The guest disables TPT every time it enters kernel-mode and re-enables it when exiting back into user-mode by writing into TPT-cr3 (similar to existing PTI logic [58]). Note that the regular cr3 register always points to the non-TPT page table, in case the hypervisor forcibly disables TPT (see §3.4 and §4.3).

To synchronize the dual page tables, the guest patches the page table operations via a new pv_ops backend when it detects TPT support. pv_ops is an existing Linux kernel API that abstracts core kernel operations in a guest OS to work optimally across different hypervisors, and is used by default on

[1] We modify the kernel's iomap to support cacheable accesses with the right PAT [56] memory attributes. Note that the virtual device's contents are backed by host memory.

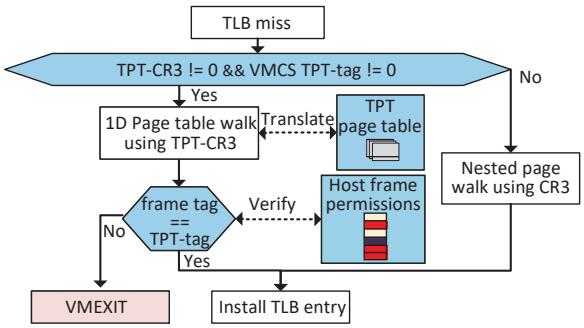

Figure 5: MMU logic to select TPT/non-TPT translation

most VMs. These core operations include page table manipulations, and the overhead of enabling this API is negligible – Linux leverages dynamic code patching to optimize calls to the hypervisor backend selected at boot time.

Every page table modification on a vanilla Linux guest uses GVAs and GPAs and goes through the `pv_ops` mechanism. TPT's `pv_ops` backend performs all requested changes as usual on the non-TPT page tables, but also synchronizes changes to user-accessible addresses with the corresponding TPT page table. Maintenance of dual page tables is therefore transparent to the guest kernel, and requires no more than 3 additional memory accesses to update the TPT page tables. First, the necessary GPA-to-HPA translation is retrieved from the guest address map. Next, the TPT page table entry is located by accessing the extended `mapping` field in `struct page` of the non-TPT page table. Finally, the TPT page table entry is updated with its new HPA value.

In most cases, the guest OS accesses a GPA before mapping it into a page table (e.g., when zeroing it), ensuring that a translation is available in its guest address map. In the few cases in which a GPA is not allocated to a HPA, the guest touches the page to force its presence, going through the pre-existing guest physical memory page-in logic of the hypervisor.

Huge page support. TPT supports huge page translation optimizations, which are used if a page is huge on both the guest and the host (note that the same happens with nested and shadow paging). Our backend adds 12 `pv_ops` functions to support TPT with huge pages.

4.3 MMU extensions

Fig. 5 shows how the hardware extensions for TPT work on a TLB miss. If `TPT-cr3` is supplied by the guest OS and the VMCS `TPT-tag` has not been zero-ed by the hypervisor (disabling TPT), the MMU uses the TPT approach for translation; otherwise, it falls back to using the VM's regular page table walk, e.g., via nested paging, as shown in the figure.

With TPT, the MMU takes each address in the page table from `TPT-cr3` as a HPA. The MMU obtains the HPA's assigned tag from the host frame permission table. It extracts the frame number from the HPA and uses it to index into the permission table. If the retrieved value matches the VMCS field `TPT-tag` (cached in an internal register), the MMU continues to the next page table level; otherwise, it raises a `TPT-fault` exception in the hypervisor. If the HPA falls into any of the VMCS `TPT-IORR` ranges, the MMU considers the HPA valid before accessing the host frame permission table.

Note that various micro-architectural optimizations are possible to perform host frame permission checks during TPT page table traversals, which are discussed in §5.

5 Discussion

Next, we discuss design alternatives, the limitations of the design, and how different design choices relate to them.

5.1 TPT with Xen-PV

TPT is not based on Xen-PV [74] due to performance and compatibility considerations.

To update guest page tables, Xen-PV employs a mechanism called direct-paging that exposes the GPA-to-HPA mappings per guest, similar to TPT's guest address map. However, unlike the TPT model, Xen-PV guests must perform costly hypercalls to update their own page tables. Xen-PV also requires all guest code to execute in ring-3, which introduces hypercall and VM traps to execute privileged guest instructions. Furthermore, Xen-PV exposes a *machine-wide* HPA-to-GPA mapping to *all of its guests*, which is required for page table management operations, and thus reduces inter-VM isolation.

From a compatibility perspective, TPT's KVM-based design is non-disruptive and allows gradual adoption: TPT's hypervisor supports both TPT and non-TPT guests, and TPT guests can run on non-TPT hypervisors (without TPT's benefits). TPT supports full hardware-based virtualization, and provides a modular and adaptable implementation.

5.2 Impact of TPT on kernel-mode

Our guest OS prototype uses TPT page tables during user-mode execution only. This is because Linux kernel-mode accesses have very small translation overheads: it maps all kernel memory using 1 GiB pages [28], making TPT's benefits in kernel mode marginal. In addition, kernel-mode TPT support would be more complex and intrusive: e.g., kernel mode has a linear map for the entire physical address space, and handles physical addresses, such as DMA and contiguous memory allocator (CMA), making the addition of two physical addressing modes (GPA and HPA) more cumbersome. We leave the exploration of kernel-mode TPT to future work.

We quantify the potential impact of kernel-mode support for TPT by measuring the performance of randomly accessing 100 GiB of memory in kernel-mode. We compare shadow paging (with direct GVA-to-HPA translations) and nested paging (with 1 GiB guest kernel pages) and confirm that TPT would provide limited benefits: nested is only 9% and 3% slower than shadow paging when using typical 4 KiB and 2 MiB host pages, respectively, whereas we see overheads of 1.5×–2.5× with the same random access pattern in a user-space application (which does not use 1 GiB pages).

5.3 Impact of TPT on memory de-duplication

By using a single, per-VM tag, TPT cannot support memory de-duplication across TPT processes on different VMs (e.g., via Linux KSM [9]). The same problem exists in AMD SEV-SNP, but is less severe in TPT because only TPT-enabled processes are subject to this limitation: all other processes and kernel-mode pages can still benefit from KSM. It would also be possible to change the hardware to assign multiple `TPT-tag` values to a single VM, and use tag mismatch exceptions to soft-multiplex a larger number.

5.4 Host frame permission check performance

TPT uses per-VM tags to check host frame permissions, something that is inspired by AMD's SEV-SNP [60]. TPT's hardware extensions are feasible with minor changes to AMD's SEV-SNP (described §4.3). Similar changes can be applied to other existing and upcoming architectures that support efficient host memory tag checks such as CHERI [72], RISC-V with PMP [71], and Arm [10, 33].

A naive MMU implementation would perform page table walks and host frame permission checks in sequence, issuing up to 9 memory accesses – a 2.7× improvement over the 24 accesses of nested paging. In practice, there are several micro-architectural optimizations to hide tag accesses and checks: (i) partial walk caches will skip intermediate host frame permission checks; (ii) tags can be embedded into the data they describe to avoid accesses; (iii) tags can be cached to reduce access times; and (iv) host frame permission accesses and checks can be overlapped with page table traversal. More specifically, AMD SEV-SNP embeds host frame tags into the data they tag within the cache hierarchy (reducing the number of accesses), and caches the tag table's contents in the regular cache hierarchy (reducing access latency). Note that some MMU implementations already use the L2$ to cache page table entries, and others have evaluated using a separate tag cache [33]. An optimized MMU implementation would hide permission check accesses by executing them in parallel to page table traversal, which can continue speculatively.

5.5 Security considerations

A malicious or faulty guest in TPT could produce page tables in which any of their levels point to an HPA not assigned to the executing VM (defined as an "incorrect HPA" from here on). An instruction that accesses memory through an incorrect HPA is never committed, but speculatively executing page table walks and permission checks in parallel could lead to potential side-channel attacks, where the page walker logic is used to prime cache lines not assigned to the executing VM.

Single-VM case. We can terminate any such VM that accesses incorrect HPAs, thus avoiding data leakage within a multi-core VM (e.g., to prime/probe cache lines separately).

Inter-VM case. Leakage could happen across colluding VMs: one VM may use the page walker to prime a cache line based on confidential data, and the other VM uses a prime/probe side-channel attack based on the line primed by the first VM [43, 78]. This is a super-set of the single-VM case above.

Such inter-VM side-channels already exist in current systems, since the micro-architectural mechanisms are the same. We can use VM termination, together with existing mitigations to resolve them, and therefore enable aggressive host frame permission check implementations: (1) immediate VM termination ensures that a channel has minimal bandwidth, and an operator can throttle VM creation when frame tag mismatch exceptions rise; (2) integrating VM tags into the cache hierarchy, as done by SEV-SNP, links permission checks and cache accesses, reducing the window of vulnerability to a single memory access (every page table walk access is tag-checked when the cache line is loaded); and (3) existing mitigation techniques are applicable to TPT, such as cache partitioning [42], or controlling the flow of micro-architectural information during speculative execution [36, 76].

Note that the guest address map exposes HPAs set by the hypervisor in response to memory usage of all VMs. This could be used as a side or covert channel between VMs, but the same is valid for memory ballooning or guest physical memory paging. The same, existing mitigations should be applied in all three cases, such as event frequency modulation.

6 Evaluation

Our evaluation demonstrates the performance gains of TPT over traditional virtualization mechanisms. To evaluate our TPT prototype, we conduct a functional emulation of its hardware capabilities on a commodity x86–64 machine, and assume that VMs only map and access their assigned pages.

Our evaluation platform cannot enforce frame tag checks in hardware, and we instead model the performance impact of the frame tag check hardware for two extreme points in the micro-architectural implementation space of the MMU (see *TPT-opt* and *TPT-naive* below). Using this approach, we assess the end-to-end impact of the proposed approach on large-scale workloads, since traditional CPU simulators would make their evaluation unfeasible. While AMD SEV-SNP already implements frame tag checks, we were unable to repurpose it to more directly evaluate TPT; this is because frame tag checks are coupled with nested paging and memory encryption, both of which introduce substantial overheads that we could not isolate.

6.1 Experimental methodology

Testbed. We use a server with 2× Intel Xeon Silver 4216 CPU and 512 GiB of memory (2× 256 GiB DDR4 2,933 GHz). It has an SR-IOV capable NIC (Mellanox ConnectX-4 Lx), which exposes dedicated virtual functions (VFs) for the host and VMs. Intel virtualization support (`VT-x`) and Intel virtualization for direct I/O access (`VT-d`) are enabled for VMs to have direct access to their dedicated VFs (using `vfio-pci` pass-through). Hyperthreading is disabled, the frequency governor is set to "performance", and "turbo" is disabled for

stable results. Both VMs and hypervisor run Ubuntu 20.04 with Linux kernel 5.16. VMs are managed by QEMU with KVM acceleration, have 16 vCPUs with 156 GiB of memory, and each vCPU is pinned to a separate physical core.

We use NUMA node 0 to evaluate both native and virtualized executions. NUMA node 1 executes client agents (without virtualization) for workloads that require requests over the network, which are passed to the physical NIC, and routed via the NIC's internal switch to the correct VF.

Hardware emulation and performance modeling. We emulate the proposed hardware extensions for TPT on the x86-64 platform in two ways:

(1) Host frame tag checks. Our base results execute on the commodity machine "as-is" and assume an optimized MMU implementation where host frame permission checks have no performance impact (*TPT-opt* below). This is a reasonable approximation as SEV-SNP has micro-architectural optimizations to hide the latency of host frame permission accesses (see §5.4), whereas traversal and check operations can be executed in parallel without compromising security (see §5.5).

We also model the performance of a naive MMU implementation where traversals and checks execute sequentially (*TPT-naive* below), resulting in an extra memory access on every page-walk cache miss. We obtain a conservative performance estimate of the naive MMU implementation by placing TPT page tables on a different NUMA node from the one with executing cores; this effectively doubles the access latency, from 81 ns to 161 ns, respectively, according to MLC [69]. A similar technique was used to model larger latencies in prior works [40,77]. This is a reasonable approximation on existing hardware since SEV-SNP avoids tag accesses by extending cache lines, and hides tag accesses by caching them (others have also proposed separate caches to avoid capacity conflicts [33]). Note that we cannot model the added memory bandwidth consumed by tag accesses, but other tagged systems show overheads below 2% on most applications, and as low as 8% in the worst case [33].

(2) MMU walker logic. Current platforms lack the necessary hardware for registers `cr3` and `TPT-cr3` and the additional MMU logic we propose to manage them, as shown in Fig. 5. We therefore emulate these extensions in software; we modify the hypervisor to intercept `cr3` operations in TPT processes, select between `cr3` or `TPT-cr3`, and enable EPT or TPT translation modes, respectively.

We add support in KVM for per-vCPU EPT control, and patch the guest OS PTI [58] assembly thunks to perform the following hypercalls when executing TPT processes:

(i) Kernel-to-user: disable EPT on the vCPU, set the guest's `cr3` to `TPT-cr3`, intercept and emulate guest `cr3` reads to return the guest's original `cr3` value (sometimes performed in exception handling during guest execution).

(ii) User-to-kernel: enable EPT on the vCPU, restore the guest's original `cr3` value, and disable `cr3` read interception.

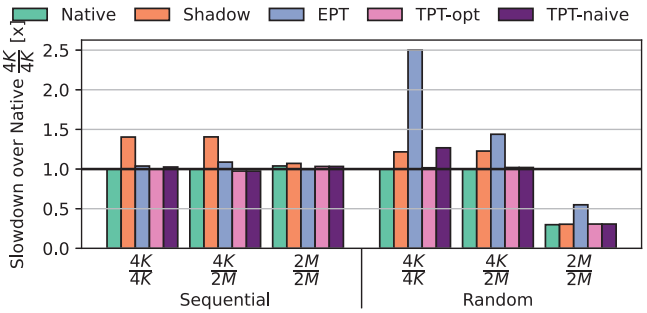

Figure 6: Relative slowdown for different translation mechanisms over native $\frac{4K}{4K}$ in memory access micro-benchmark (Lower is better.)

With a hardware implementation, the decision to use `cr3` or `TPT-cr3` would be performed by hardware with negligible cost. However, our software emulation has additional overheads for performing the additional hypercalls (via `VMCALL` and `VMEXIT`), which may dominate execution time on system call- or interrupt-heavy workloads (PTI is also only enabled for TPT).

We therefore report execution times after subtracting the software emulation overheads (time spent on new hypercalls performing EPT and `cr3` manipulations) from the application execution time. Note that most benchmarks do not invoke the hypercalls during the evaluation phase, and thus we do not remove the emulation overhead in such cases. For experiments that need the emulation hypercalls, such as the page table manipulation micro-benchmarks, we configure them to execute in a single physical CPU to maintain modeling correctness.

Configurations. We evaluate TPT's performance against the following system configurations:

(a) Native: Native execution, which serves as our ideal, upper bound on performance.

(b) Shadow: VM with shadow paging.

(c) EPT: VM with nested paging using Intel's extended page tables (EPT) mechanism.

(d) TPT-opt: VM with an optimized TPT implementation.

(e) TPT-naive: VM with a naive TPT implementation.

We evaluate each of the configurations under different host/guest page sizes: (1) $\frac{4K}{4K}$: guest OS uses base pages (4 KiB), and host backs VM with base pages (4 KiB); (2) $\frac{4K}{2M}$: guest OS uses base pages (4 KiB) and host backs VM with huge pages (2 MiB); and (3) $\frac{2M}{2M}$: guest uses huge pages (2 MiB) by enabling transparent huge pages (THP), and host backs VM with huge pages (2 MiB).

6.2 Memory translation

We create a single-thread memory micro-benchmark to assess TPT's impact on memory performance under different scenarios. Our benchmark allocates a 100 GiB buffer to serve as the target for memory read operations at a 64-bit granularity.

First, we evaluate the performance of sequential and ran-

Shadow	EPT	TPT-opt	TPT-naive
18.8×	5×	6.68×	6.68×

Table 2: Relative slowdown for different translation mechanisms over native $\frac{4K}{4K}$ when manipulating the guest page table using `mprotect` (Lower is better.)

dom memory accesses. Fig. 6 shows the relative slowdown of the evaluated system configurations over the native execution time. TPT-opt's performance matches that of Native, regardless of the memory page size, under both sequential and random access patterns. This is expected, as TPT-opt eliminates all virtualization-induced memory translation overheads due to its use of direct GVA-to-HPA page tables.

We also observe that EPT has the worst performance under the random access pattern due to its two-dimensional page walk on each TLB miss. Huge pages reduce, but do not eliminate the overhead of EPT, as the page walk is shortened due to the smaller size of the two-dimensional page walk [47].

TPT-naive only underperforms in random memory accesses under the $\frac{4K}{4K}$ configuration, where it exhibits a 1.25× relative slowdown over TPT-opt and Native. This is the result of the longer page walk duration on TLB misses, but it is still 2× faster than EPT under the same configuration.

Shadow exhibits a slowdown of 1.41× and 1.25× over Native for sequential and random memory accesses, respectively. The overheads stem from the extra time spent in the hypervisor due to page faults, which cause expensive VMEXITs. Although the guest OS page tables are populated, the shadow page tables are maintained by the host and updated lazily and on-demand: accesses to newly-mapped pages incur a page fault that is handled by the hypervisor, which in turn updates the shadow page tables and resumes guest execution.

Conclusions: TPT-opt exhibits native performance, outperforming both Shadow and EPT.

6.3 Page table management overheads

Raw page table manipulation. We evaluate the performance of page table modification under all configurations, as each incurs overheads from different sources. We measure the time taken to downgrade a single page from read-write to read-only via the `mprotect` system call.

Table 2 shows the results, normalized to Native execution. Shadow exhibits a slowdown of more than 18× over Native, as downgrading permissions in the page tables require TLB invalidations that are trapped by the host to amend the shadow page tables. EPT does not require interventions by the host and incurs a slowdown of 5× for a single page permission modification. This is the result of a single TLB entry invalidation in the guest (using instruction `INVLPG`), which invalidates *all* paging-structure translation caches of the current context, including the partial-walk caches (PWC) [8, 32, 47]. We corroborate this finding by observing an increase in the number of cycles the hardware page table walkers are active under the

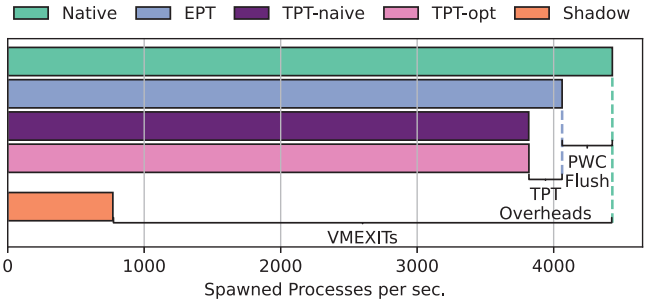

Figure 7: Spawn micro-benchmark for different translation mechanisms (Higher is better.)

Workload	Description	RSS
kcbench	Kernel compilation benchmark (v4.19) [35]	1 GiB
XSBench	Monte Carlo neutron transport algorithm [65]	99 GiB
Canneal	Optimization for chip design (PARSEC [20])	109 GiB
GUPS	Random integer updates in memory (HPCC [44])	129 GiB
PR	Page Rank (GAPBS [17]) on kron graph[2]	72 GiB
BFS	BFS Algorithm (GAPBS) on kron graph	70 GiB
CC	CC Algorithm (GAPBS) on kron graph	70 GiB
Memcached	Facebook ETC [11] (3×10^8 keys; mut. client [39])	108 GiB

Table 3: Application workloads and memory footprint

EPT configuration over Native (not shown).

TPT-opt exhibits a small overhead over EPT due to the additional operations to keep dual page tables in sync. Note that TPT-opt is also subject to the same cache invalidation overheads triggered by `INVLPG`, and TPT-naive has no additional overheads because page table contents are always accessed via non-TPT page tables in kernel space.

We perform the same experiment to measure the performance of mapping anonymous memory, by evaluating the `mmap` system call with the `MAP_POPULATE` flag. We do not observe a performance difference between the configurations, as the majority of time is spent on physical memory allocations and zeroing page contents.

TPT's extra logic to manipulate dual page tables has a small performance impact, and does not affect the end-to-end results on our evaluated applications (see §6.4). Such low overheads to synchronize modifications across page tables are corroborated by prior work [3, 53].

Process spawning. We evaluate the performance of Spawn from the Unixbench benchmark suite [67]. Spawn is a fork/wait-type workload, which measures the number of times a process can fork and reap a child that immediately exits.

Fig. 7 shows that EPT, TPT-opt, and TPT-naive are subjected to the adverse effects of PWC flushes as the `fork` system call performs TLB invalidations. The additional operations in TPT (maintaining dual page tables) lead to a 1.06× slowdown over EPT. In comparison, shadow incurs a 5.18× slowdown over EPT due to `VMEXIT`s induced by TLB invalidation and page faults.

[2] Kron graph [38] scale: 2^{29}, average degree: 16

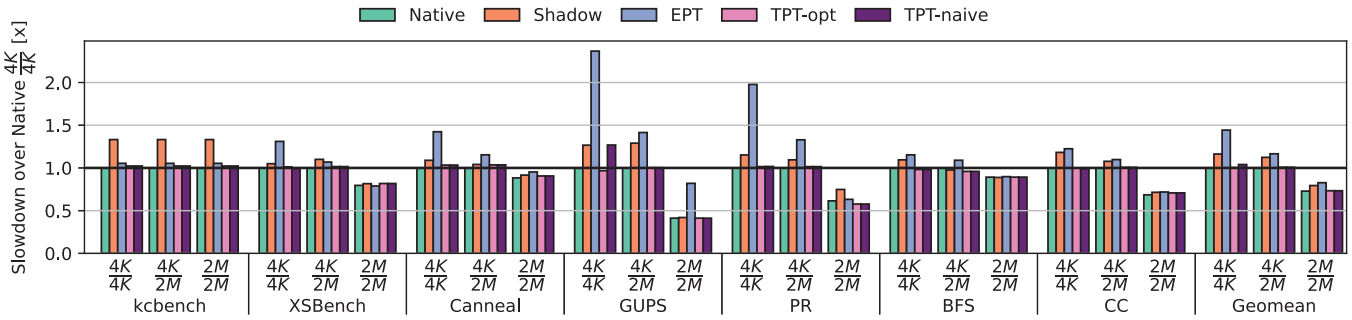

Figure 8: Relative slowdown on applications benchmarks over native $\frac{4K}{4K}$ (Lower is better.)

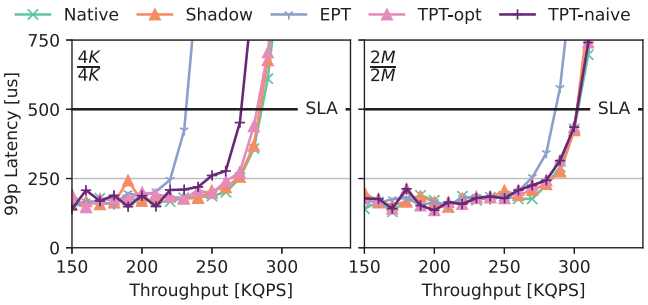

Figure 9: Memcached throughput-latency for different translation mechanisms (Lower is better.)

Conclusions: TPT enables substantially faster page table manipulations compared to Shadow by eliminating `VMEXITs`. It exhibits higher page table manipulation costs over EPT and native, but they have negligible impact on end-to-end application performance, as shown in §6.4.

6.4 Application benchmarks

We evaluate TPT on the application benchmarks listed in Table 3, which are commonly used to evaluate of data centers workloads [27, 53]. All benchmarks, apart from Memcached and kcbench, execute with 8 threads. We execute Memcached with a single thread because we do not have enough client cores to saturate more server threads. For maximum performance, Memcached uses the VMA [41] library for user-level I/O. We evaluate kcbench with a single thread because we need to model performance in the face of emulation hypercalls (as explained in §6.1).

Fig. 8 shows the relative slowdown of all the evaluated configurations over Native $\frac{4K}{4K}$. TPT-opt matches the performance of Native for all workloads, under all page size configurations. EPT exhibits significant slowdowns in workloads with random memory accesses, such as GUPS and PR. Overheads for Shadow are apparent in workloads that perform memory mappings, such as process spawning (kcbench), and dynamic memory allocation with page table modifications (CC and GUPS). TPT-naive exhibits a slowdown only on GUPS with the $\frac{4K}{4K}$ configuration, of 1.25×. GUPS is a random memory access benchmark, which correlates with our previous results for the random access micro-benchmark in Fig. 6. The geometric mean of the slowdown for TPT-naive is of just 3%.

Huge pages reduce the virtualization overheads of both EPT and Shadow, as well as improve the performance of Native execution, because they reduce TLB misses and page walk costs. Huge pages substantially decrease overheads in EPT as they reduce the number of steps on each dimension of the walk. EPT, however, still exhibits noticeable slowdowns with huge pages over TPT-opt and Native. Shadow's overheads with huge pages decrease, as 2 MiB mappings, compared to 4 KiB ones, induce less `VMEXITs` to sync the shadow page tables with the guest's page table mappings.

Memcached. We single out the Memcached benchmark, because it is latency-sensitive. Fig. 9 shows the throughput-latency graph of the 99th percentile of Memcached serving Facebook's ETC requests, with an SLA of 500 µs (following previous work [18]). Although the ETC access distribution is skewed, the keys are small in size and randomly distributed. This affects the overall access distribution of the workload, which exhibits a random memory access pattern.

EPT performs the worst due to the random memory accesses. Shadow, Native, and TPT-opt perform similarly in both page size configurations. This is expected because no new memory allocations occur during the measured portion of the workload. TPT-naive under the $\frac{4K}{4K}$ configuration crosses the SLA with 4% lower throughput than TPT-opt. The mean latency exhibits the same behavior as the 99th percentile, although the knee of the curves occurs at a higher throughput.

Conclusions: The performance of TPT matches Native, and systematically outperforms Shadow and EPT on all benchmarks where page table management or memory access performance dominates, respectively, even with huge pages.

6.5 Impact of 1 GiB huge pages

We now evaluate the same applications in §6.4 with EPT $\frac{2M}{1G}$, using 1 GiB host pages (typically unfeasible in a production system). The applications in Fig. 8 with 1 GiB pages only show a 2.5% speedup (geometric mean) compared to our previous EPT $\frac{2M}{2M}$ results, which would correspond to a 1.16× slowdown over Native and TPT-opt. In turn, Memcached's throughput only increases by 2.5% with 1 GiB pages, which

corresponds to a 1.04× slowdown over TPT-opt and Native.

Conclusions: Unlike TPT, 1GiB huge pages do not eliminate nested paging overheads completely.

6.6 Memory overheads

Guest. Non-TPT processes have no memory overheads, but TPT processes incur a small overhead to hold the TPT page tables. The TPT page tables only map the user space pages of the process and do not map the entire system memory and kernel space. A process with a sequential memory mapping of n pages incurs an additional $(1-\lfloor\frac{\text{page\_size}}{2\,\text{MiB}}\rfloor)\cdot\lceil\frac{n}{2^{36}}\rceil+\lceil\frac{n}{2^{27}}\rceil+\lceil\frac{n}{2^{18}}\rceil+\lceil\frac{n}{2^{9}}\rceil$. For example, a mapping of 1 GiB with 4 KiB pages incurs an extra 2 MiB-worth of TPT page tables.

Host. TPT's guest address map requires 8 B of host memory per GFN to hold the HFN. Therefore, a 64 GiB VM consumes 256 KiB or 128 MiB of host memory if the host utilizes 2 MiB or 4 KiB pages respectively (less than 0.2% in both cases).

Conclusions: TPT only adds small memory overheads in guests and hosts, making it practical for adoption.

7 Related Work

Prior work either attempts to improve existing virtualization mechanisms [27, 31, 46, 50, 53, 70], thus inheriting their shortcomings, or introduces invasive hardware changes, potentially changing the behavior of VMs compared to native execution [5–7, 15, 19, 23, 45, 55, 62].

Hardware-based virtualization. DVMT [6] proposes a software MMU architecture where applications/VMs can use their own address translation structures. DVMT also uses tag-based frame protection for isolation, but retains a two-dimensional translation approach in VMs, albeit with customizable translation structures on each dimension.

Sha et al. [19] propose new paging schemes for processors with software MMUs. The schemes reduce the page walk cost in nested paging by incorporating flat nested page table [5], or reduce the cost of updating guest page tables in shadow paging by intercepting TLB flushes. However, it introduces a software MMU and constraints to guest physical address space size, making it difficult to apply to modern machines and large memory sizes, respectively.

Several studies propose to redesign paging structures. Compromis [23] uses direct segments, but requires the reservation of variable-length physical memory areas for segments, which significantly compromises the flexibility of memory management in hypervisors. Chang et al. [55] propose to flatten 2 levels of page tables to reduce the cost of page walks by half. This approach increases the cost and complexity of managing page tables, and its cost reduction is limited. Nested Elastic Cuckoo Page Tables [62] utilize hashed page tables to reduce the nested page walk cost. However, replacing the existing radix page tables with hashed page tables requires significant changes to existing software and hardware ecosystems.

Caching and prefetching are also effective at hiding translation latency. Thomas et al. [15] explored new MMU caches, including today's partial walk caches in AMD and Intel processors. ASAP [45] reduces address translation latency by storing multiple page tables contiguously and introducing a hardware prefetcher for page walks. Caching does not fully eliminate the memory translation costs, and prefetching can result in mispredictions and numerous memory accesses to the page table when accessing large virtual memory areas. MMU caches and prefetching are directly applicable to TPT.

Improving virtualization. Agile paging [27] combines shadow and nested paging to reduce the hypervisor intervention cost on page table updates. It requires more memory accesses per TLB miss than native machines and TPT.

On-demand virtualization [31] enables virtualization dynamically to migrating bare-metal machines, and disables virtualization after the migration. This approach only applies to bare-metal machine migration, and cannot be generalized: it does not support more than a single VM in a bare-metal machine, and cannot enforce isolation between the VM and the hypervisor, because it relies on identity mappings (1:1) for nested translation between the VM and the hypervisor.

8 Conclusions

TPT is a new approach to memory virtualization, which achieves near-native translation performance for memory-intensive applications in VMs. In TPT, VMs regain control over their translation structures, while maintaining memory isolation across VMs by leveraging emerging physical memory protection technologies. TPT is compatible with both TPT and non-TPT guests, and can be selectively applied to processes running within any TPT-aware VM.

Acknowledgements

We gratefully acknowledge support from the Israel Science Foundation (grants 980/21 and 1027/18). This work was also partially supported by the Technology Innovation Institute (TII) through its Secure Systems Research Center (SSRC), and by JSPS KAKENHI grant number 18KK0310 and JST CREST grant number JPMJCR22M3, Japan.

References

[1] 5-Level Paging and 5-Level EPT. Technical report, Intel Corp., December 2016.

[2] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3), 2006.

[3] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth*

International Conference on Architectural Support for Programming Languages and Operating Systems, pages 283–300, 2020.

[4] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for X86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 2–13, New York, NY, USA, 2006. Association for Computing Machinery.

[5] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 476–487, 2012.

[6] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 457–468, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 515–528. IEEE, 2020.

[8] AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf. Last accessed: December 2022.

[9] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, pages 19–28. Citeseer, 2009.

[10] Arm. *Armv8.5-A Memory Tagging Extension*, August 2019.

[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[12] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System Performance Evaluation of Para-virtualization, Container Virtualization, and Full Virtualization using Xen, Openvz, and Xenserver. In *2014 Fourth International Conference on Advances in Computing and Communications*, pages 247–250. IEEE, 2014.

[13] S Anish Babu, MJ Hareesh, John Paul Martin, Sijo Cherian, and Yedhu Sastri. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 fourth international conference on advances in computing and communications*, pages 247–250. IEEE, 2014.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[15] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 48–59, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Thomas W Barr, Alan L Cox, and Scott Rixner. SpecTLB: A Mechanism for Speculative Address Translation. *ACM SIGARCH Computer Architecture News*, 39(3):307–318, 2011.

[17] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619*, 2015.

[18] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[19] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, 2008.

[20] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. "the parsec benchmark suite: Characterization and architectural implications". In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[21] CLOC: Count Lines of Code. https://github.com/AlDanial/cloc. Last accessed: December 2022.

[22] Confidential Computing Consortium. https://confidentialcomputing.io/. Last accessed: December 2022.

[23] Boris Teabe Djomgwe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, and Gilles Muller. (No) Compromis: Paging Virtualization Is Not a Fatality. In *VEE 2021-17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–12, 2021.

[24] Yaozu Dong, Jinquan Dai, Zhiteng Huang, Haibing Guan, Kevin Tian, and Yunhong Jiang. Towards High-Quality I/O Virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–8, 2009.

[25] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[26] Wei Fan and Albert Bifet. Mining Big Data: Current Status, and Forecast to the Future. *ACM SIGKDD Wxplorations Newsletter*, 14(2):1–5, 2013.

[27] Jayneel Gandhi, Mark D Hill, and Michael M Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 707–718. IEEE, 2016.

[28] x86, mm: Enable GBPAGES Option by Default. https://github.com/torvalds/linux/commit/9e899816d126cc6f7d405c349f65363214fe7399. Last accessed: December 2022.

[29] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[30] Irfan Habib. Virtualization with KVM. *Linux Journal*, 2008(166):8, 2008.

[31] Jaeseong Im, Jongyul Kim, Youngjin Kwon, and Seungryoul Maeng. On-demand Virtualization for Post-copy OS Migration in Bare-metal Cloud. *IEEE Transactions on Cloud Computing*, pages 1–1, 2022.

[32] Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide*, 2022.

[33] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W Moore, Alex Bradbury, Hongyan Xia, Robert NM Watson, David Chisnall, Michael Roe, Brooks Davis, et al. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

[34] The Operstack Foundation. Kata Containers - The Speed of Containers, the Security of VMs. https://katacontainers.io/. Last accessed: December 2022.

[35] kcbench: Linux Kernel Compilation Benchmark. https://gitlab.com/knurd42/kcbench. Last accessed: December 2022.

[36] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown with Leakage-free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[37] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual Remote I/O. *ACM SIGARCH Computer Architecture News*, 44(2):49–65, 2016.

[38] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 133–145. Springer, 2005.

[39] Jacob Leverich. Mutilate: High-Performance Memcached Load Generator, 2014.

[40] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[41] LibVMA. https://github.com/Mellanox/libvma/wiki/Architecture. Last accessed: December 2022.

[42] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418. IEEE, 2016.

[43] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622. IEEE, 2015.

[44] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE*

Conference on Supercomputing, volume 213, pages 1188455–1188677, 2006.

[45] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1023–1036, 2019.

[46] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. Ptemagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, 2021.

[47] Timothy Merrifield and H Reza Taheri. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 25–35, 2016.

[48] Microsoft Azure Confidential Computing Powered by 3rd Gen EPYC™ CPUs. https://community.amd.com/t5/business/microsoft-azure-confidential-computing-powered-by-3rd-gen-epyc/ba-p/497796. Last accessed: December 2022.

[49] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A comparison study of Intel SGX and AMD memory encryption technology. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.

[50] Jun Nakajima, Qian Lin, Sheng Yang, Min Zhu, Shang Gao, Mingyuan Xia, Peijie Yu, Yaozu Dong, Zhengwei Qi, Kai Chen, et al. Optimizing Virtual Machines Using Hybrid Virtualization. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 573–578, 2011.

[51] Neiger, Gil and Santoni, Amy and Leung, Felix and Rodgers, Dion and Uhlig, Rich. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), 2006.

[52] OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com/. Last accessed: December 2022.

[53] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized numa servers with vmitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–210, 2021.

[54] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360, 2019.

[55] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. Every Walk's a Hit: Making Page Walks Single-Access Cache Hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–141, 2022.

[56] PAT (Page Attribute Table). https://www.kernel.org/doc/html/latest/x86/pat.html. Last accessed: December 2022.

[57] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html. Last accessed: December 2022.

[59] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[60] AMD SEV-SNP. "strengthening vm isolation with integrity protection and more". *White Paper, January*, 2020.

[61] The x86 KVM Shadow MMU. https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt. Last accessed: December 2022.

[62] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 84–97, New York, NY, USA, 2022. Association for Computing Machinery.

[63] TDP MMU. https://lwn.net/Articles/832835/. Last accessed: December 2022.

[64] Intel Trust Domain Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf. Last accessed: December 2022.

[65] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[66] Ubuntu 14.04 On Amazon EC2: Xen PV vs. HVM. https://www.phoronix.com/review/amazon_ec2_pvhvm. Last accessed: December 2022.

[67] Unixbench. https://github.com/kdlucas/byte-unixbench. Last accessed: December 2022.

[68] Prashant Varanasi and Gernot Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[69] V Viswanathan, Karthik Kumar, and T Willhalm. Intel Memory Latency Checker. *Intel Corporation*, 2013.

[70] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Selective Hardware/Software Memory Virtualization. *ACM SIGPLAN Notices*, 46(7):217–226, 2011.

[71] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. *RISC-V Foundation*, 2019.

[72] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[73] Xen Project Software Overview. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. Last accessed: December 2022.

[74] Xen Project. X86 Paravirtualised Memory Management. https://wiki.xenproject.org/wiki/X86_Paravirtualised_Memory_Management. Last accessed: December 2022.

[75] Xen PV Performance Status and Optimization Opportunities. https://www.slideshare.net/xen_com_mgr/xen-pv-performance-status-and-optimization-opportunities. Last accessed: December 2022.

[76] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[77] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[78] Yuval Yarom and Katrina Falkner. FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX security 14)*, pages 719–732, 2014.