Direct Memory Translation for Virtualized Clouds

Jiyuan Zhang
University of Illinois
Urbana-Champaign, IL, USA

Weiwei Jia
University of Rhode Island
Kingston, RI, USA

Siyuan Chai
University of Illinois
Urbana-Champaign, IL, USA

Peizhe Liu
University of Illinois
Urbana-Champaign, IL, USA

Jongyul Kim
University of Illinois
Urbana-Champaign, IL, USA

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA

Abstract

Virtual memory translation has become a key performance bottleneck of memory-intensive workloads in virtualized cloud environments. On the x86 architecture, a nested translation needs to sequentially fetch up to 24 page table entries (PTEs). This paper presents Direct Memory Translation (DMT), a hardware-software extension for x86-based virtual memory that minimizes translation overhead while maintaining backward compatibility with x86. In DMT, the OS manages last-level PTEs in a contiguous physical memory region, termed Translation Entry Areas (TEAs). DMT establishes a direct mapping from each virtual page in a Virtual Memory Area (VMA) to the corresponding PTE in a TEA. Since processes manage memory with a handful of major VMAs, the mapping can be maintained per VMA and effectively stored in a few dedicated registers. DMT further optimizes virtualized memory translation via guest-host cooperation by directly allocating guest TEAs in physical memory, bypassing intermediate virtualization layers. DMT is inherently scalable—it takes one, two, and three memory references in native, virtualized, and nested virtualized setups. Its scalability enables hardware-assisted translation for nested virtualization. Our evaluation shows that DMT significantly speeds up page walks by an average of 1.58x (1.65x with THP) in a virtualized setup, resulting in 1.20x (1.14x with THP) speedup of application execution on average.

CCS Concepts: **• Software and its engineering → Operating systems**; **Virtual memory**; **• Computer systems organization → Architectures**.

Keywords: Cloud Computing, Virtualization, Virtual Memory, Address Translation

ACM Reference Format:
Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. 2024. Direct Memory Translation for Virtualized Clouds. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640358

1 Introduction

Virtual memory translation has become a major performance bottleneck of memory-intensive workloads, especially in virtualized cloud environments. Upon a TLB miss, a nested address translation on the x86 architecture needs to perform a two-dimensional page table walk, which requires up to 24 *sequential* memory accesses. It is reported that nested translation can take more than 50% of the execution time of memory-intensive workloads in virtualized environments [5, 20, 45]. With the rapid growth of memory capacity and irregular memory access patterns of emerging workloads, TLB misses are becoming more frequent, with increased overhead.

Recent support for five-level page tables in new processors [30, 81] may further slow down memory translation—a nested translation would require up to 35 sequential memory accesses. In addition, new cloud deployment modes for multi-level virtualization require higher dimensional page table walks. For example, nested virtualization [11, 28] runs hypervisors inside virtual machines (VMs). Currently, nested virtualization is supported by shadow paging on top of nested translation, bearing the known cost of frequent VM exits [1, 11]. However, extending hardware-assisted translation to three (or more) dimensions of page tables is untenable.

To address the pressing overhead of memory translation, advanced software and hardware designs are proposed [3, 4, 6, 10, 15, 17, 19, 20, 25, 26, 29, 37–40, 43–47, 55, 56, 58–61, 63, 64, 80, 84]. However, translation overhead remains substantial in virtualized environments. Existing designs that reduce page table walks (e.g., using huge pages [35, 40, 55, 56]) cannot eliminate sequential memory accesses. Translation caching and prefetching [9, 39, 44–46, 63, 84] can save (or hide) memory access overhead; however, even with perfect caching, the translation still needs to *sequentially* fetch tens of Page Table Entries (PTEs) from the cache hierarchy, which takes orders of magnitude more cycles than a TLB hit.

This paper presents Direct Memory Translation (DMT), a hardware-software extension for x86-based virtual memory that minimizes address translation overhead while maintaining backward compatibility with x86. DMT co-designs the translation hardware together with OS memory management to reduce translation overhead to what is absolutely necessary—directly fetching the translation that maps a virtual page to the corresponding physical page frame (in x86, the translation is stored in the last-level PTEs of radix page tables). DMT achieves only *one*, *two*, and *three* memory references in native, virtualized, and nested virtualized setups.

The high-level idea of DMT is to establish a direct mapping from each virtual page to the corresponding last-level PTE that stores the translation. The mapping can be effectively maintained at the granularity of Virtual Memory Areas (VMAs). VMAs are contiguous virtual memory regions in process address space; they are mostly allocated at the initialization time and are infrequently changed at runtime. DMT manages the mapping from each VMA to a contiguous physical memory region (termed TEA, or Translation Entry Area) that stores last-level PTEs of pages in the VMA in order. Similar to VMAs, TEAs are managed by the OS. Since processes manage memory with a handful of major VMAs (§2.3), the VMA-to-TEA mapping can be effectively stored in a few dedicated registers (16 in our implementation). In case that processes have more than 16 VMAs, DMT merges adjacent VMAs and/or maps the largest VMAs in the available registers. With the mapping in place, DMT directly locates the last-level PTE in a TEA for a given page in a VMA.

DMT is inherently more scalable than existing x86 radix-tree based translation design, as it directly fetches last-level PTEs, without sequential page table work or resorting to page walk caches. Note that such property also exists in hash-based translation [64–66, 69, 80]. DMT has a few advantages over hash-based designs: (1) DMT uses fixed-size TEAs instead of hash tables, which greatly simplifies the design. The rationale is that VMAs usually remain static after creation; enlarging and shrinking VMAs that require costly TEA modifications are rare. Hash-based designs, on the other hand, cannot know the number of PTEs apriori and thus need to resize the hash tables as PTEs are added or removed; (2) DMT avoids hash collisions [80] and/or reduces parallel lookups [64, 66]; and (3) Although both TEAs and hash tables need contiguous physical memory, DMT is more flexible because it can split its tables when contiguous memory is unavailable or fall back to the baseline design.

We further optimize DMT for virtualized environments. Specifically, DMT employs guest-host cooperation (aka paravirtualization) to directly allocate guest TEAs in physical memory—guest TEAs always reside in a contiguous physical memory region of the host. We refer to the optimized design as pvDMT, which bypasses intermediate virtualization layers and thus accelerates multi-dimensional page walks. Compared to hashed page tables [80], pvDMT reduces the number of sequential steps from 1, 3, and 7 to 1, 2, and 3 in native, virtualized, and nested virtualized setups, respectively. To ensure isolation and prevent side channels, pvDMT restricts that a guest can only map its virtual pages to its own TEAs (instead of arbitrary physical addresses), with a mechanism similar to Intel EPTP switching [32]. As DMT scales linearly with the levels of virtualization, it enables hardware-assisted translation for nested virtualization.

We prototype DMT based on the x86 architecture and Linux/KVM. DMT requires simple hardware support that equips each processor core with 16 registers and simple PTE fetch logic. The hardware support is similar to ASAP PTE prefetcher [45], which is backward compatible with x86 and thus considered practical. DMT supports all existing virtual memory features, such as huge pages and page sharing, and is transparent to x86 binaries. We evaluate DMT with a range of data-intensive workloads and compare DMT with four other advanced designs, including FPT [59], ECPT [64, 66], Agile Paging [20], and ASAP [45]. Our results show that pvDMT can effectively speed up page table walks by 1.58x (1.65x with THP) in a virtualized environment, resulting in 1.20x (1.14x with THP) speedup of workload execution on average; it substantially outperforms other advanced designs like ECPT [66] by 1.16x–1.31x (1.25x–1.51x with THP) in a virtualized environment. DMT's performance gains on nested virtualization are more substantial.

In summary, the paper makes the following contributions:

- Direct memory translation (DMT), a practical hardware-software extension for x86-based virtual memory to minimize address translation overhead. DMT co-designs the translation hardware and OS memory management so that sequential page table walks are replaced by directly fetching last-level PTEs.
- An optimization of DMT in virtualized setups using paravirtualization that further reduces translation overhead to two memory references for virtualization and three for nested virtualization.
- An implementation of DMT on x86 and Linux/KVM. The DMT prototype and other artifacts are available at: https://github.com/xlab-uiuc/dmt.

2 Background

2.1 Address Translation in Virtualized Clouds

In this section, we provide a brief overview of the *existing* page table structure, the architectural support for virtualized memory, and the design for nested virtualization.

2.1.1 Radix Page Tables. Radix page table is the *de facto* design of most current architectures, which organizes the page table entries (PTEs) in a multi-level radix tree. Currently, the x86-64 architecture implements a 4-level tree. To translate a memory address upon a TLB miss, the memory management unit (MMU) walks over each level of the tree

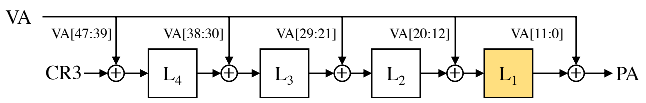

Figure 1. Address translation in a native environment. DMT directly fetches the last-level entry (highlighted).

sequentially, as shown in Figure 1. Therefore, in the worst case, a page table walk requires four sequential memory accesses. x86-64 supports huge pages of 2MB and 1GB. For huge pages, the translation is shortened by one or two levels.

To accommodate emerging terabyte-scale memory capacity, Intel extends the radix page table to five levels [30, 81], which further increases the page-table walk cost.

2.1.2 Virtualized Memory Translation. Virtualization significantly enlarges memory translation overhead. A guest OS manages its page table independently from the host (the hypervisor) and runs on virtualized physical memory. Therefore, a *guest virtual address* (gVA) needs to first be translated into a *guest physical address* (gPA) which is further translated into a *host physical address* (hPA).

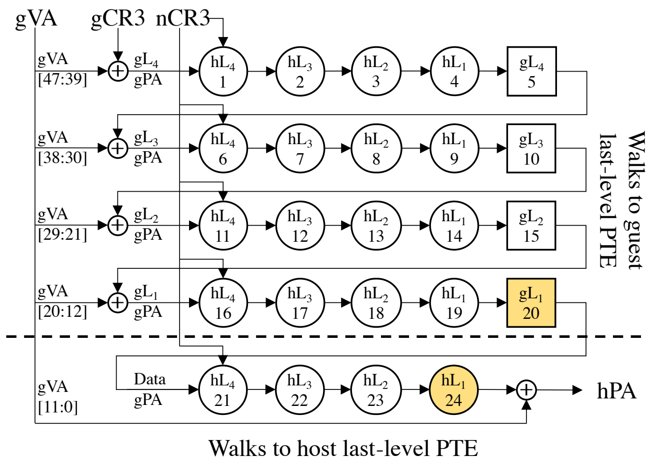

Figure 2. Two-dimensional (2D) address translation in a virtualized environment. DMT aims to fetch only two translation entries (highlighted).

Modern architectures support hardware-assisted virtualized memory translation, aka *nested paging* (e.g., Intel EPT [32] and AMD NPT [16]). Nested paging uses two layers of page tables: each guest OS maintains a *guest page table* (gPT) that maps gVAs to gPAs, and the hypervisor manages a *host page table* (hPT) per guest that maps gPAs to hPAs. To translate a gVA to hPA upon a TLB miss, the hardware issues a two-dimensional (2D) page table walk, as illustrated in Figure 2. We use circular boxes to denote levels of the host page table (hL_i) and square boxes to denote levels of the guest page table (gL_i). First, to obtain the gPA from gVA, the hardware needs to walk over each level of the gPT, from gL_4 to gL_1. To access each gL_i, the hardware needs to know the hPA of gL_i by referring to the hPT, from hL_4 to hL_1. The above process corresponds to Steps 1–20 in Figure 2. Second, to obtain the hPA from the gPA, the hardware needs to further walk over the hPT (Steps 21–24 in Figure 2). For 4-level radix page tables, the 2D page table walk requires up to 24 memory references, as shown in Figure 2. With 5 levels, it takes up to 35 memory references.

Even when MMU caches and huge pages are used, the 2D page table walks remain expensive and can take more than 50% of the execution time of memory-intensive workloads [5, 20, 45].

An alternative to hardware-assisted translation is shadow paging [20, 74], where the hypervisor builds a *shadow page table* (sPT) by combining gPT and hPT. While a guest OS manages gPT, the hypervisor maintains the sPT to map gVA to hPA, and the translation takes a native page walk (§2.1.1). However, any update of the gPT must be synchronized with the sPT, causing frequent VM exits [1, 2]. It is known that shadow paging incurs substantial overhead, e.g., it is reported that hardware-assisted translation speeds up applications up to 48% over shadow paging [72].

2.1.3 Nested Virtualization. Nested virtualization [11] runs hypervisors inside virtual machines (VMs), creating *multiple* levels of virtualization. It is widely offered by cloud vendors [7, 14, 82] and enables many use cases [11, 28, 42, 83]. For example, Windows 10/11 runs a type-1 hypervisor (Hyper-V) for kernel integrity [50] and Linux interoperability [49]; therefore, nested virtualization needs to be supported for Windows 10/11 on a VM [50, 73].

Currently, no architectural support is available for memory translation of nested virtualization. One main reason is that, with the current x86 design, nested virtualized memory translation would require a three-dimensional (3D) page table walk, which has an untenable overhead.

Now, nested virtualization is implemented by mapping the three levels of page tables onto the two levels of nested paging (§2.1.2), where the page tables of the two hypervisors are compressed into one sPT, as shown in Figure 3. The $\mathbb{L}_0$ hypervisor maintains an sPT that maps $\mathbb{L}_2$PA to $\mathbb{L}_0$PA, combined from $\mathbb{L}_1$PT and $\mathbb{L}_0$PT. To translate an $\mathbb{L}_2$VA into $\mathbb{L}_0$PA needs a 2D page table walk across $\mathbb{L}_2$PT and sPT, so that the $\mathbb{L}_2$VA is translated to $\mathbb{L}_2$PA, then to $\mathbb{L}_0$PA. The implementation uses a combination of nested paging and shadow paging to accommodate three page tables. Hence, it suffers from the overheads caused by both.

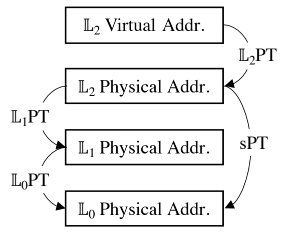
$\mathbb{L}_2$ Virtual Addr.
$\mathbb{L}_2$PT
$\mathbb{L}_2$ Physical Addr.
$\mathbb{L}_1$PT
$\mathbb{L}_1$ Physical Addr.
sPT
$\mathbb{L}_0$PT
$\mathbb{L}_0$ Physical Addr.

Figure 3. Address space and page tables in nested virtualization.

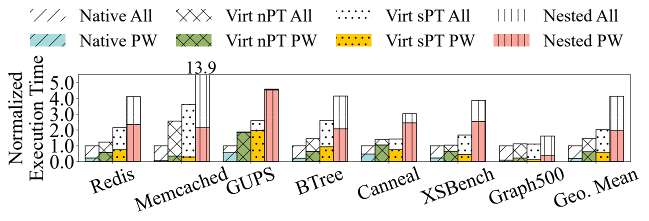

Figure 4. Execution time of seven benchmarks under (1) native, (2) virtualized (with nested paging), (3) virtualized (with shadow paging), and (4) nested virtualized environments. The numbers are normalized to the execution time of the native environment. The page-table walk overhead is highlighted.

2.2 Translation Overhead in Virtualized Clouds

Figure 4 shows the address translation overhead of native, virtualization (including both nested paging and shadow paging), and nested virtualization on an x86 server machine, respectively, under various workloads. The measurement methodology and the machine configuration are detailed in §5. Compared with the native environment, virtualization increases the execution time by 1.46x on average, due to the overhead of a two-dimensional (2D) page table walk (§2.1.2). Nested virtualization further increases execution time by 4.13x on average. Specifically, on average, the page table walk overhead of native, virtualization, and nested virtualization are 21%, 43%, and 48%. Nested virtualization incurs an additional overhead of sPT (§2.1.3).

To quantify the sPT overhead, we compare the execution time in the virtualized environment, using hardware-assisted nested paging and shadow paging, respectively. On average, shadow paging increases the total execution time by 1.39x compared to nested paging, even though the page table walk is simplified (28% page table walk overhead).

In short, address translation overhead in virtualized environments, including both single-level and nested virtualization, is excessive for memory-intensive workloads. The overhead is expected to be higher with 5-level page tables.

2.3 Virtual Memory Area

VMA is an OS abstraction of contiguous regions in the virtual address space of a process. Each VMA contains a set of virtual pages with the same protection, representing a local data section (e.g., code, data, heap, stack, or a memory-mapped file) [22]. It has a base virtual address and size of the mapped region, along with other metadata. Collectively, VMAs of a process constitute the application's working set [25].

There are typically a few hundred VMAs in a modern process. On the other hand, it is reported that only a handful of them are of significant size and are frequently accessed [25, 45]. To validate this observation, we measure VMA characteristics of different applications as well as the SPEC CPU 2006 and 2017 benchmarks. Table 1 and Figure 5 show the measured VMA characteristics. Our results validate the reported observation in most workloads. In all workloads except Memcached and two workloads in the SPEC CPU 2017 benchmark (548.exchange2_r and 648.exchange2_s), 16 VMAs cover 99% of the working set. We observe that the heap VMA covers most of the working set. The two SPEC CPU 2017 workloads have 20 and 21 VMAs respectively, while Memcached has 778 VMAs. Despite the large number of VMAs, we observe that those VMAs are close to each other in the virtual address space, with small bubbles in between (e.g., less than 16KB in Memcached). If we cluster adjacent VMAs with an allowance of 2% of bubbles in total, 99% of the working set can be covered by less than 12 VMA clusters in *all* the workloads. Hence, DMT works at the granularity of VMAs or VMA clusters.

Table 1. VMA characteristics of various workloads: "Total" number of VMAs, the number of VMAs that cover 99% of Total ("99% Cov."), and the number of VMA "Clusters" with small bubbles of less than 2% in total to cover 99% of Total (Memcached's 778 VMAs can be covered by two clusters).

Workload (WL)	Total	99% Cov.	Clusters
BTree	109	2	2
Canneal	116	2	2
Graph500	105	1	1
GUPS	103	1	1
Redis	182	6	6
XSBench	111	1	1
Memcached	1,065	778	2
SPEC CPU 2006 (30 WLs)	18–39	1–14	1–8
SPEC CPU 2017 (47 WLs)	24–70	1–21	1–12

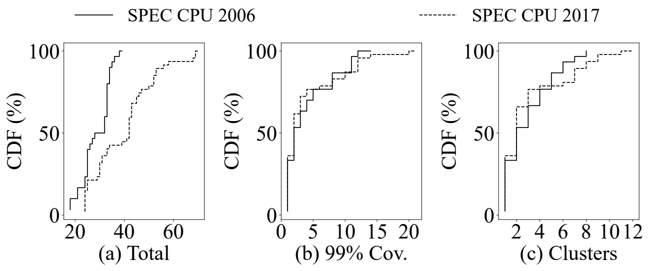

Figure 5. CDFs of the three VMA characteristics in Table 1 of SPEC CPU 2006 and 2017 benchmarks. (containing 30 and 47 workloads, respectively).

3 Direct Memory Translation

Direct Memory Translation (DMT) is a hardware-software extension for x86-based virtual memory to minimize address translation overhead while retaining backward compatibility with the x86 architecture. DMT reduces translation overhead to what is necessary—directly fetching the last-level PTEs of the x86 radix page tables without sequentially walking the page tables or resorting to page walk caches. The last-level PTE contains the translation that maps a virtual page to the

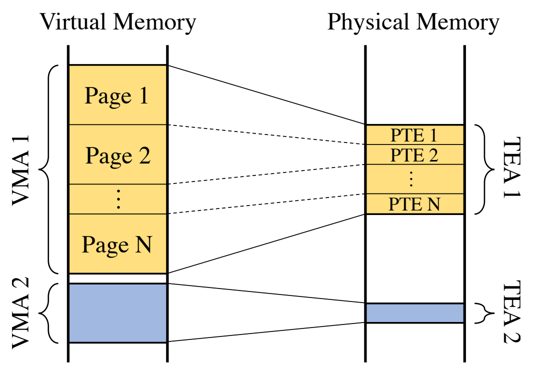

Figure 6. Basic idea of direct memory translation. Each VMA has a corresponding TEA (managed by the OS) which contains last-level PTEs of pages in VMAs in the same order.

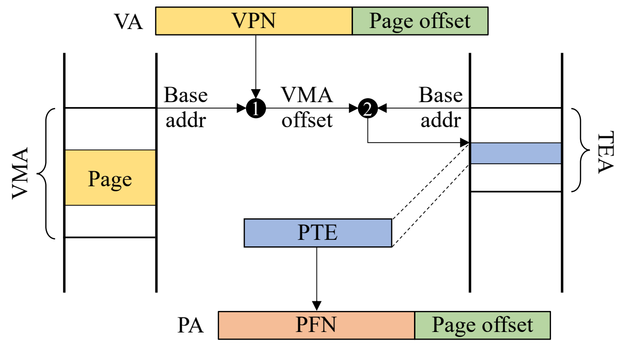

Figure 7. Direct memory translation procedure. DMT directly fetches the last-level PTE of the virtual page based on the VMA-to-TEA mapping.

physical page frame. DMT can achieve *one*, *two*, and *three* memory accesses in native, virtualized, and nested virtualized setups (with paravirtualization-based optimizations).

The high-level idea is to establish a direct mapping from each virtual page to the corresponding last-level PTE in physical memory. DMT maintains the mapping based on the VMA abstraction (§2.3), because (1) processes manage memory with a handful of VMAs (§2.3) and (2) VMAs are mostly allocated at the initialization time and are infrequently changed at runtime. Since each VMA is a contiguous *virtual* memory region consisting of multiple virtual pages, DMT maps the VMA to a contiguous *physical* memory region containing the last-level PTEs of the virtual pages in the VMA. We term the contiguous physical region *Translation Entry Area* (TEA). The size of a TEA is orders of magnitude smaller than its corresponding VMA, in the same scale as how a PTE size compares to page sizes; hence, the contiguity requirement of TEA is viable (discussed in §7). Figure 6 illustrates the direct VMA-to-TEA mappings established by DMT.

The VMA-to-TEA mapping can be efficiently stored in a few dedicated hardware registers (§4.1). These registers are a part of the task state—during a context switch, registers of the new process are reloaded. Our implementation uses 16 registers, which is sufficient for most applications (§2.3). In case processes have more than 16 VMAs, DMT either maps the largest VMAs in the available registers or clusters adjacent VMAs with bubbles in the TEA (§4.2).

With the VMA-to-TEA mapping in place, DMT directly locates the last-level PTE for a given virtual address (VA). Figure 7 illustrates the translation procedure: ❶ with the VMA base address, DMT calculates the virtual page number (VPN) offset of the VA inside the VMA; ❷ with the TEA base address, DMT indexes the target PTE using the VPN offset. As such, in the native case, DMT always takes only one memory reference to fetch the last-level PTE.

A conscious design principle of DMT is to simplify hardware translation by co-designing OS memory management for TEAs. This principle makes DMT backward-compatible with x86 and thus avoids disruptive changes to MMUs, page tables, and virtual memory abstractions. The design makes the right tradeoffs: (1) VMAs mostly remain static after creation; enlarging and shrinking VMAs that require OS involvement for TEA modifications are rare; (2) the OS is in the right position to dynamically merge or split TEAs, based on memory contiguity or other policies.

Conceptually, DMT resembles the translation using a linear page table [41] which stores all PTEs in a single array indexed by the VPN, where each TEA is analogous to a linear page table of the VMA. Managing TEAs at the granularity of VMAs solves the space inefficiency of traditional linear page tables that map to the entire process address space. Compared to hash-based translation, the DMT design greatly simplifies the translation hardware; for example, DMT avoids hash collisions [80] and parallel lookups [64]. Moreover, it is compatible with the current x86 architecture and can always fall back to the default translation mechanism.

Note that DMT does not create additional copies of PTEs. Hence, PTE access and dirty bits work in the same way as the existing system, with no race condition. For the same reason, it does not incur TLB shootdown due to TEA invalidations. DMT does not prefetch PTEs either. Prefetching is less effective in virtualized environments. A large number of PTEs may cause cache contention. More importantly, the translation still needs to fetch many PTEs from the cache hierarchy *sequentially* with substantial overhead (§6.2.2).

3.1 DMT for Virtualization

DMT can be directly applied to virtualized environments, as shown in Figure 8. DMT would take three memory references to translate a guest virtual address (gVA) to the host physical address (hPA). The translation procedure is as follows (we use prefixes "g" and "h" to refer to the guest and the host, respectively): (1) DMT calculates the guest physical address (gPA) of the last-level guest PTE (gPTE) based on the gVMA-to-gTEA mapping; it then translates the gPA to the hPA of the gPTE using the hVMA-to-hTEA mapping based on

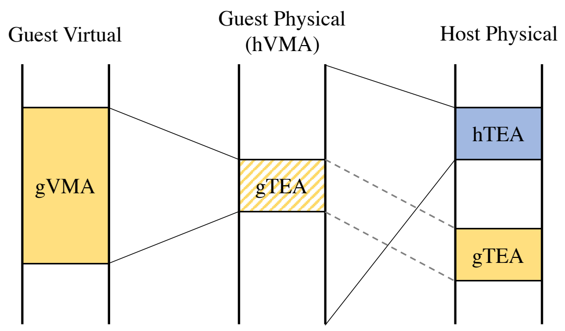

Figure 8. DMT for virtualized environments (solid line: VMA-to-TEA mapping; dashed line: VM address mapping). With pvDMT, gTEAs are contiguous in the *host* physical memory space and do not need to be contiguous in the *guest* physical memory space.

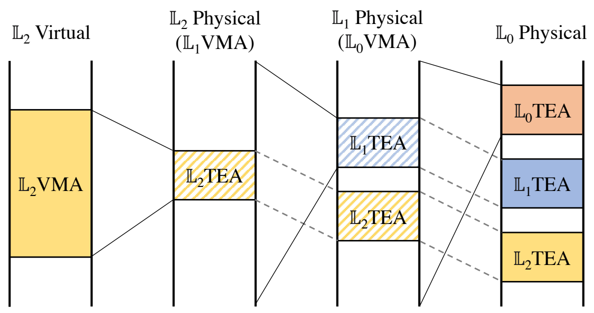

Figure 9. DMT for nested virtualization (solid line: VMA-to-TEA mapping; dashed line: VM address mapping). With pvDMT, $\mathbb{L}_1$TEAs and $\mathbb{L}_2$TEAs are contiguous in $\mathbb{L}_0$ physical memory space and do not need to be contiguous in the $\mathbb{L}_1$ and $\mathbb{L}_2$ physical memory space.

the host PTE (hPTE); (2) DMT fetches the gPTE in physical memory and obtains the gPA of the target data page; and (3) DMT finds the hPA of the target data page using the corresponding hPTE via the hVMA-to-hTEA mapping.

The above procedure achieves the best-case scenario as hash-based designs [66, 80] without hash collisions and parallel lookups, e.g., Nested ECPT [66] also take three sequential accesses but multiple (up to 81) parallel ones.

Paravirtualization. We further optimize DMT for virtualized environments to only *two* memory references with guest-host cooperation (aka paravirtualization). The optimized DMT, named pvDMT, allocates gTEAs directly in the host physical memory—gTEAs always reside in contiguous host physical memory regions managed by the hypervisor. With pvDMT, the gVMA-to-gTEA mappings are built directly from gVA to hPA, bypassing the indirection in the guest physical address space; the host maintains hVMA-to-hTEA mappings in the same way as in the native case (Figure 6). An hVMA is the hypervisor's VMA corresponding to the guest physical address space of a virtual machine. Note that pvDMT does not require gTEA to be placed in contiguous *guest* physical memory space.

Essentially, pvDMT fetches only two PTEs: (1) the last-level gPTE that maps gVA to the gPA via the gVMA-to-gTEA mapping, and (2) the last-level hPTE that maps the gPA to the hPA via the hVMA-to-hTEA mapping. These two PTEs are the ones highlighted in Figure 2. pvDMT brings substantial performance advantages over the state of the art in virtualized environments, where PTEs are harder to cache compared with the native environment.

Note that pvDMT needs to be carefully designed to ensure isolation. Our design restricts that a guest can only map its virtual pages to its own TEAs using a mechanism similar to Intel EPTP switching, eliminating side channels (see §4.5.2).

3.2 DMT for Nested Virtualization

The idea of DMT (including pvDMT) can further be applied to nested virtualization, as shown in Figure 9. In the case of pvDMT, in nested virtualization, the TEAs at $\mathbb{L}_2$, $\mathbb{L}_1$, and $\mathbb{L}_0$ are all allocated in contiguous host physical memory regions. As a result, pvDMT takes at most three memory references to translate an $\mathbb{L}_2$VA to $\mathbb{L}_0$PA, with the following procedure: (1) DMT fetches the $\mathbb{L}_2$PTE based on the $\mathbb{L}_2$VMA-to-$\mathbb{L}_2$TEA mapping and then translates $\mathbb{L}_2$VA to $\mathbb{L}_2$PA; (2) DMT fetches the $\mathbb{L}_1$PTE based on the $\mathbb{L}_1$VMA-to-$\mathbb{L}_1$TEA mapping and then translates $\mathbb{L}_2$PA to $\mathbb{L}_1$PA; and (3) DMT obtains $\mathbb{L}_0$PA.

Since pvDMT scales linearly with the levels of virtualization, it can enable hardware-assisted address translation for nested virtualization. The hardware-assisted translation could eliminate the major overhead of nested virtualization caused by shadow paging for synchronizing the $\mathbb{L}_1$ and $\mathbb{L}_0$ page tables (§2.1.3). Note that hardware-assisted translation for nested virtualization is untenable in traditional radix-tree based design or the recent ECPT design, both of which takes an excessive number of memory accesses for single-level virtualization already (24 sequential accesses with radix page tables and 81 parallel accesses with Nested ECPT).

4 Design and Implementation

DMT can be practically supported on modern architectures and operating systems with simple hardware extensions. The hardware extension includes a set of registers for maintaining VMA-to-TEA mappings and simple translation logic in MMU for fetching last-level PTEs (§4.1). The hardware extension is similar to the one in the ASAP prefetcher [45].

The main effort of enabling DMT is to design and implement the OS support for (1) dynamically managing the VMA-to-TEA mappings (§4.2), (2) effectively managing TEAs (§4.3), (3) supporting modern virtual memory features such as huge pages (§4.4), and (4) virtualization support and the

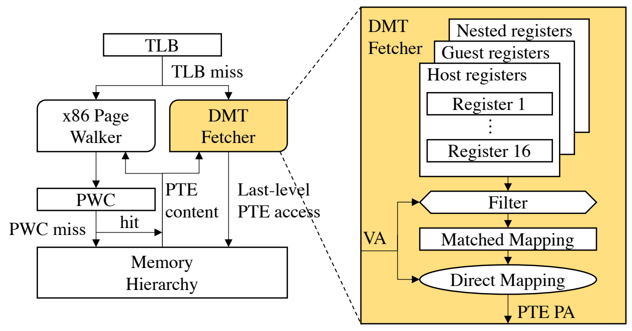

Figure 10. Overview of the DMT hardware extension.

paravirtualization-based optimization (pvDMT) for both single-level and nested virtualization (§4.5). We prototype the OS support for DMT on top of the Linux kernel v5.15.0, referred to as DMT-Linux (§4.6). Our experience shows that DMT-Linux can be practically implemented.

4.1 Hardware Extension

Figure 10 depicts the hardware extension required by DMT, named DMT Fetcher. A DMT fetcher includes a set of dedicated registers and fast translation logic. It co-exists with the current x86 page table walker. The DMT fetcher is compatible with the x86 architecture and works with existing architecture structures. The TLB, x86 page table walker, and MMU caches (e.g., page walk caches) are untouched.

The DMT fetcher maintains a number of registers (16 in our implementation, §5). Each register stores a VMA-to-TEA mapping. These registers are available for each hardware thread and are exposed to the OS as part of the task state. The registers are updated by the OS on events like context switches and interrupts in virtual machines.[1]

Upon a TLB miss, a page table walk request is handled by the DMT fetcher or falls back to the x86 page table walker, depending on whether the requested address is covered by the mappings in the registers. If the corresponding VMA-to-TEA mapping is present in a register, the DMT fetcher directly locates the last-level PTE (§3). We expect that 99+% of the page table walk requests are served by the DMT fetcher because a handful of VMAs or VM clusters typically cover 99% of the memory footprint of data-intensive workloads (see §2.3). DMT dynamically clusters adjacent VMAs for workloads that have large numbers of VMAs (§4.2).

Different from ASAP [45] that prefetches the last *two levels* of PTEs into the cache hierarchy, DMT locates only the last-level PTEs. It does not bring upper-level PTEs into CPU caches and thus consumes less memory bandwidth.

The hardware support for virtualization (and nested virtualization) shares the same principles (described in §4.5).

[1]When KVM handles external interrupts, VM exits may be triggered to switch from the guest to the host. The registers need to be reloaded [32]. On a multi-core system, IPIs are required to update registers on remote cores.

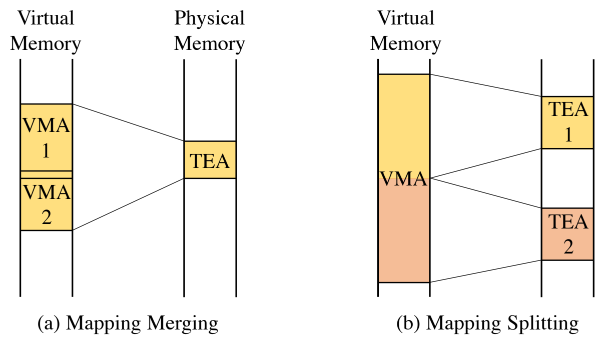

Figure 11. The VMA-to-TEA mapping can be dynamically merged (§4.2.1) and splitting (§4.2.2).

4.2 VMA-to-TEA Mapping Management

The OS (i.e., DMT-Linux) manages the VMA-to-TEA mappings and stores them in the DMT registers (§4.1). For each process, DMT-Linux sorts the VMAs in descending order based on their sizes and stores mappings that cover the largest VMAs in the registers. DMT-Linux prioritizes storing large VMAs in registers rather than hot VMAs. Large VMAs are typically for heap and memory-mapped regions that are the primary causes of page table walks. Small but hot VMAs mostly represent dynamically linked libraries and the stack, which are frequently accessed and rarely cause TLB misses due to high temporal locality [45].

Managing VMA-to-TEA mappings needs to address the following challenges: (1) there can be many more VMAs (e.g., Memcached in Table 1) than the number of DMT registers, so per-VMA mapping may not reach a high coverage; (2) given that TEA requires contiguous memory space, it could be difficult to allocate a large TEA in highly fragmented memory; and (3) mapping management needs to accommodate potential changes of VMAs.

4.2.1 Merging VMA-to-TEA Mappings. DMT-Linux may merge mappings of closely located VMAs to increase the memory coverage of the registers, as shown in Figure 11(a). Upon creating a new mapping, DMT-Linux checks whether the target VMA can be clustered with the adjacent VMAs. DMT-Linux uses a configurable threshold t to make the decision based on the ratio of the bubbles (t is set to 2% by default). If the bubble ratio resulting from the merging is calculated to be below t, DMT-Linux clusters the two VMAs and merges their mappings. This process is performed iteratively until the ratio is larger than t. The merging involves TEA expansion (§4.3) that first tries to expand a TEA in place. If successful, the other TEA is migrated to the expanded TEA. If the in-place expansion fails, it allocates a new TEA and migrates the original TEAs to it.

4.2.2 Splitting VMA-to-TEA Mapping. The size of a TEA is orders of magnitude smaller than the size of the corresponding VMA (a 4KB page of TEA covers 2MB VMA). However, an allocation of TEA for a large VMA or VMA cluster can fail due to highly fragmented memory. If the memory allocator fails to allocate a TEA, DMT-Linux splits the TEA using multiple mappings to cover the VMA, as shown in Figure 11(b). Specifically, it uses two mappings, each corresponding to half of the original VMA. The splitting iterates until the TEA allocation succeeds. Note that after splitting, the VMA can be removed from the hardware register and be replaced by a larger VMA.

4.2.3 Accommodating VMA Changes.

A VMA can grow or shrink during the process execution. For example, `mmap` can grow an existing VMA and `munmap` can shrink it. When a VMA grows, DMT-Linux expands the size of the mapped TEA accordingly (§4.3). The TEA expansion is triggered on VMA changes. If a VMA is shrunk, DMT-Linux reduces the size of the corresponding TEA. As the size of a VMA gets smaller, its mapping in a register can be replaced by the mapping of a larger VMA.

Note that VMA operations are typically infrequent. In our evaluation, the workloads rarely change VMAs after the initialization. In principle, DMT trades the overhead of infrequent VMA-to-TEA mapping management for optimizing frequent virtual address translations.

4.3 TEA Management

TEAs are managed by the OS (DMT-Linux). To support TEAs, we modify the page table allocator in Linux. Currently, in Linux, PTEs are maintained in pages allocated by the buddy allocator [22] and last-level PTEs are randomly scattered in the physical memory. DMT-Linux incorporates a specialized page table allocator that allocates TEAs in physical memory based on the VMA-to-TEA mappings. It allocates PTEs in specific locations inside the TEAs to enable DMT.

DMT-Linux supports TEA creation, deletion, and expansion. A TEA is created upon a VMA-to-TEA mapping creation. A TEA is allocated using Linux's contiguous physical page allocator (`alloc_contig_pages`) which requests contiguous pages from the buddy allocator. The request could fail if no contiguous region could be allocated, which triggers a split of the VMA-to-TEA mapping. DMT-Linux also instructs the memory allocator to defragment the memory to resolve moveable fragmentations. A TEA is deleted by simply freeing it upon VMA deletion.

TEA expansion is triggered either by the merging events of VMA-to-TEA mappings (§4.2.1) or by organic VMA growth (§4.2.3). If a TEA is expandable in place in physical memory, DMT-Linux allocates PTEs into the corresponding TEA. Since the PTEs are allocated in the page granularity by the Linux buddy allocator, DMT-Linux manages TEAs also in the page granularity. If a TEA cannot be expanded in place, DMT-Linux creates a new TEA and copies the original TEA to the new one. DMT-Linux implements gradual migration: the new TEA will be loaded into the register so the expanded VMA can be indexed properly, while PTEs are asynchronously moved by a background thread. During the TEA migration, translations with PTEs that are not yet migrated will be handled by the original x86 page table walker.

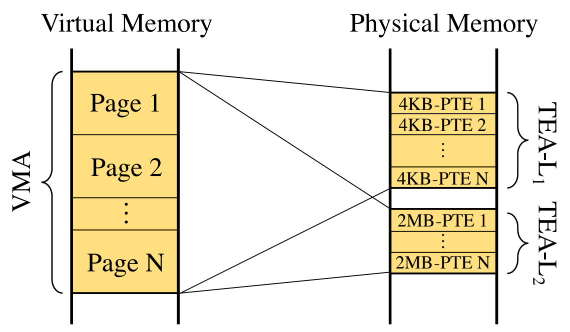

Figure 12. Huge page support in DMT.

4.4 Huge Page Support

DMT supports huge pages like 2MB and 1GB pages in the x86 architecture. With huge pages, last-level PTEs are at L_2 and L_3, not L_1 (Figure 1). If a VMA contains pages of multiple sizes, DMT maps the VMA to multiple different TEAs, each containing last-level PTEs of different sizes of pages, as depicted in Figure 12. For example, if a VMA contains pages of all three sizes, there will be three VMA-to-TEA mappings for 4KB, 2MB, and 1GB pages, respectively. To translate a VA in such a VMA, the DMT fetcher issues three memory requests in parallel to all TEAs. Since only one of the three TEAs contains the last-level PTEs, only one PTE will be fetched. The page size is encoded in the SZ field of the register when storing the VMA-to-TEA mapping (Figure 13). The DMT fetcher calculates the location of the target PTEs inside a TEA based on the page size.

When a huge page is promoted or demoted (e.g., by Linux's Transparent HugePage Support, or THP), the corresponding VMA-to-TEA mapping does not need to be changed. Only the PTEs in the TEA will be updated. PTEs of different sizes of pages are allocated in different TEAs.

4.5 Virtualization Support and pvDMT

DMT for virtualized memory can be supported naturally: the OS (DMT-Linux) support in §4.2–§4.4 works seamlessly on both the guest and the host OSes. The hardware support includes a new set of registers that map the gVA of guest VMAs to the gPA of the corresponding guest TEA (gTEAs). As discussed in §3, without paravirtualization (i.e., pvDMT), the DMT fetcher finishes the translation with three memory references: (1) it calculates the gPA of the target gPTE using the gVMA-to-gTEA mapping in the guest DMT registers and then translates the gPA to the hPA of the gPTE via the hPTE using the hVMA-to-hTEA mapping in the host DMT registers; (2) it fetches the gPTE and obtains the gPA of the target data page; and (3) it finds the hPA of the target data page via the hPTE using the hVMA-to-hTEA mapping in the host DMT registers. In practice, the hypervisor typically creates one VMA to represent the guest physical memory.

4.5.1 Paravirtualization. To support the paravirtualization-based optimization (pvDMT), we need to ensure gTEAs to be contiguous in the host physical memory (§3.1). In pvDMT, the host allocates gTEAs for the guest and maps the allocated gTEAs into the guest so that the guest can update PTEs without exiting the VM. Guest TEA (gTEA) operations, such as merging and splitting VMA-to-TEA mappings, thus need to be forwarded to the host. In this way, only one VM exit would occur when a TEA is created or updated.

DMT-Linux adds a new hypercall `KVM_HC_ALLOC_TEA` in the guest OS to pass VMA-to-TEA mapping information to the host. The host prepares contiguous *physical* memory regions for gTEAs and, if needed, merges or splits the mapping. It then maps the materialized mapping to the guest. The use of a hypercall not only enables the guest to request the host to allocate gTEAs but also allows the host to merge or split gTEAs when the requested allocation cannot or should not be satisfied as is. The hypercall takes an array of *requested* gTEAs as input parameters and returns an array of *allocated* gTEAs, in the form of a pointer to a gTEA Table that lists the base addresses and sizes of gTEAs. It returns an empty array if no TEA can be allocated.

The gTEA table is maintained by the host, as part of the register state (Figure 13). It lists the base address and size of each gTEA in the host physical address space. The table is needed for the DMT fetcher to directly fetch the gPTE.

With pvDMT, the DMT fetcher takes two memory references: (1) it fetches the gPTE to obtain the gPA of the target page in the gTEA using the gVMA-to-gTEA mapping in the guest DMT register, where the base address of gTEA in the host physical address space can be found in the gTEA Table, and (2) it fetches the hPTE to obtain the physical address based on the gPA in the hTEA using the hVMA-to-hTEA mapping in the host register.

4.5.2 Isolation. A key principle of pvDMT is to ensure isolation between the guest and the host. The challenge is that paravirtualization exposes the host physical memory to the guest. Without any regulation, a malicious guest can create a timing-based side channel by manipulating the guest DMT register to point to an arbitrary host physical address. Since the MMU will read the content at the physical addresses and consume it as a PTE, the guest can observe the timing of the translation fault and infer the shape of the content in the physical memory address. On the other hand, VM exits need to be minimized—regulation cannot require every gTEA change or context switch to trap into the host.

We address the challenge by restricting the physical memory region exposed to the guest and ensuring that a guest can only map its virtual pages to its own TEAs. The restriction is realized by the gTEA table using a mechanism similar to Intel EPTP Switching [32]. Basically, the host creates a gTEA table for each guest, which tracks the physical memory regions of the gTEAs that belong to the guest VM. The gTEA table is read-only to the guest; any gTEA modification must go through the `KVM_HC_ALLOC_TEA` hypercall. The gTEA table maintains the base address in the host physical memory and the size of every active gTEA for the currently running guest VM. Each gTEA is assigned a unique ID stored in the guest registers, as shown in Figure 13. The base address of the gTEA table is also stored in the guest registers. Note that this design also prevents the host physical address of TEAs from being exposed to the guest.

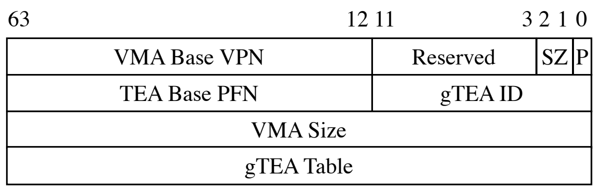

Figure 13. Organization of a DMT register. SZ stands for "page size" and P stands for "present". gTEA ID and gTEA Table are used by pvDMT specifically.

During the translation, the DMT fetcher finds the corresponding gTEA based on the gTEA table and the gTEA ID from the guest DMT register. If a given gTEA ID is invalid or an out-of-bound physical memory access is requested, a page fault will be triggered in the host.

4.5.3 Nested Virtualization. DMT and pvDMT enable efficient hardware-assisted address translation for nested virtualization, which is untenable with existing translation designs (§2.1.3). In the case of pvDMT, to support memory translation for nested virtualization, we need a set of $\mathbb{L}_2$ VMA-to-TEA registers that map the $\mathbb{L}_2$ VA to the host physical address of the corresponding $\mathbb{L}_2$ TEA. The DMT fetcher needs to support the three-step translation described in §3.2.

DMT-Linux makes the $\mathbb{L}_1$ hypervisor be able to identify the destination of control messages and forward them between $\mathbb{L}_0$ and $\mathbb{L}_2$ if needed. For example, after $\mathbb{L}_1$ DMT-Linux received the TEA allocation request from $\mathbb{L}_2$, it will create and map TEAs for $\mathbb{L}_2$ on $\mathbb{L}_1$ physical memory, split the mapping if needed, and forward the updated VMA-to-TEA mappings to $\mathbb{L}_0$ for final allocation. The allocated mappings are returned from $\mathbb{L}_0$ to $\mathbb{L}_1$ and then to $\mathbb{L}_2$.

4.6 Implementation

4.6.1 DMT Hardware. Figure 13 shows the layout of a DMT register (§4.1). We have three sets of 16 registers to support native, virtualization, and nested virtualization, respectively. Each register maintains a VMA-to-TEA mapping using the first 192 bits. The registers can only be accessed by the corresponding native or virtualization levels.

The P (Present) bit notifies the DMT fetcher if the VMA-to-TEA mapping stored in the register is valid. If the P-bit is not set, the original x86 page table walker is invoked. For example, during the asynchronous TEA migration (§4.3), the P-bit is not set till the migration is done.

The gTEA ID and gTEA Table fields are specific to pvDMT and are only used in virtualized environments. The gTEA ID is used by the guest to point to the gTEA in the gTEA table; the gTEA table is prepared by the host. The fields may be changed during context switches, without VM exits.

Following the current MMU design, a core can have multiple DMT fetchers corresponding to the original x86 page table walkers. If a requested virtual memory address is not covered by the mappings, the translation falls back to the x86 page table walker. For virtualized memory, the DMT fetcher performs the multi-step translation described in §4.5, following a state machine to access the right registers.

4.6.2 DMT-Linux.

We prototype DMT-Linux on top of the Linux kernel v5.15. We implemented VMA-to-TEA management, TEA management, and virtualization support with about 800 lines of C code.

To manage VMA-to-TEA mappings (§4.2), we hook our management procedures into code that changes VMAs (e.g., `mmap_region`, `__vma_adjust` and `__split_vma`). The mappings are maintained in a red-black tree with pointers in `task_struct`, similar to how VMAs are maintained.

TEAs are allocated using Linux contiguous page allocator, `alloc_contig_pages`, implemented in the buddy allocator with `GFP_PGTABLE_USER` (DMT is not enabled for the kernel page table). The allocator automatically defragments memory on demand. DMT-Linux invokes `__alloc_contig_pages` and `free_contig_range` to expand and shrink TEAs. To migrate a TEA, a background worker iterates through the page table using `apply_to_existing_page_range`, locking the affected PTEs, and revising the page table to ensure the original page walker works correctly after migration.

To support DMT for virtualization (§4.5), DMT-Linux adds the hypercall `KVM_HC_ALLOC_TEA` in KVM. Upon receiving the hypercall, the hypervisor finds guest physical memory in the host virtual address using `kvm_vcpu_gfn_to_hva` and `find_vma`. It allocates TEAs on behalf of the guest and maps the allocated host physical memory to the guest physical memory using `vm_insert_pages`. After the mapping is done, DMT-Linux updates the gTEA table and the gTEA ID (Figure 13) and returns the allocated mappings to the caller. In nested virtualization, it cascades the hypercall and requests the $\mathbb{L}_0$ hypervisor for the same operations.

Hardware extension (§4.1), if available, can be supported by using `mm_struct` to store per-process mappings and procedures like `switch_mm` to load mappings to registers.

Table 2. Hardware configuration of the measurement platform (a server machine with the x86 architecture).

Parameter	Configuration
Processor	Intel® Xeon® Gold 6138 @ 2.00GHz (4 CPU sockets)
Memory	64GB DDR4 2666 MT/s per socket (256GB total)
Host Kernel	Linux 5.15.0
Guest Kernel	Linux 5.15.0
Hypervisor	QEMU/KVM
VM Memory	240 GB

Table 3. Configuration of the simulated architecture. We set the same hardware configuration as the measurement machine in Table 2, i.e., Intel Xeon Gold 6138 [76].

Parameter	Configuration
Cores/Threads	20 cores; 20 threads
L1I TLB	128 entries, 8-way associative
L1D TLB	64 entries, 4-way associative
L2 STLB	1536 entries, 12-way associative
L1I Cache	32KB per core, 8-way associative, 4 cycles RT
L1D Cache	32KB per core, 8-way associative, 4 cycles RT
L2 Cache	1MB per core, 16-way associative, 14 cycles RT
Last-Level Cache	22MB, 11-way associative, 54 cycles RT
Main Memory	200 cycles RT
Page Walk Cache	3 levels, 2-4-32 entries per level, 1 cycle RT
Nested PWC	3 levels, 2-4-32 entries per level, 1 cycle RT

5 Methodology

We evaluate the performance of DMT and the compared systems, following the methodology of prior work [5, 45]. We model application execution time using a combination of real performance measurements and simulated memory traces. The execution time on a *target* system is modeled as:

$$T^{target} = O^{measure}_{vanilla} \times \frac{O^{simulate}_{target}}{O^{simulate}_{vanilla}} + T^{measure}_{ideal}$$

where O denotes address translation overhead. T_{ideal} denotes the ideal execution time assuming a perfect TLB. $O^{simulate}_{vanilla}$ and $O^{simulate}_{target}$ denote the translation overhead of vanilla Linux and the target system in a simulator, respectively. The formula breaks down application execution time into an ideal execution time and the page table walk overhead. As suggested by [5], it is more accurate than directly adding simulated translation overhead on measured execution time, by taking into account subtle interactions between the translation hardware, CPU cores, and the OS.

We use the Linux Perf tool [52] to measure $O^{measure}_{vanilla}$ and workload execution time of a real server machine, with the configuration listed in Table 2. $T^{measure}_{ideal}$ is obtained by subtracting translation overhead from total execution time. To measure $O^{simulate}_{vanilla}$ and $O^{simulate}_{target}$, we use DynamoRIO [18] to simulate the memory hierarchy. Table 3 lists the configuration of the simulated system. We implement different translation designs including DMT and the related systems on the simulator. The simulator runs the application memory

Table 4. Benchmarks used in the evaluation.

Name	Description
Redis [62]	In-memory key-value store (working-set: 155 GB) 512M 256B records, 30M operations, 100% reads
Memcached [48]	In-memory key-value store (working-set: 95 GB) 100M 1KB records, 10M operations, 100% reads
GUPS [27]	Random memory accesses (working-set: 128 GB) 128 GB dataset, 1B operations, 100% updates
BTree [51]	Data structure evaluator (working-set: 125 GB) 1.5B keys, 70M operations, 100% lookups
Canneal [13]	Chip design optimizer (working-set: 62 GB) 100M elements, 150K swaps/step, 2K start temp.
XSBench [71]	Monte Carlo simulator (working-set: 84 GB) 170K gridpoints per nuclide, 4M particle histories
Graph500 [24]	Graph analysis benchmark (working-set: 123 GB) 27 scale, 32 edge factor, 4 iterations

traces ($\geq$ 2 billion instruction level trace) and measures the simulated overheads. The simulator also reports the average page table walk latency of the target systems.

For nested virtualization, since DMT eliminates the need for shadow paging, we need to remove the shadow paging overhead (O^{nested}_{shadow}) from T_{total}. However, it is hard to precisely measure O^{nested}_{shadow}. Inspired by [20], we run the benchmarks with hardware-assisted translation and with shadow paging on single-level virtualization and measure $T^{measure}_{total}$ respectively. The difference of the measured T_{total} tells the overhead of shadow paging over hardware-assisted translation. Since shadow paging overhead is mainly caused by VM exits, we estimate the overhead in nested virtualization by scaling the overhead measured in single-level virtualization based on the ratio of VM exits, i.e., $O^{nested}_{shadow} = O^{single}_{shadow} \times \frac{N^{nested}}{N^{single}}$. Note that this is an underestimation because VM exits are known to be more expensive in nested virtualization [11].

Benchmarks. Our evaluation uses a diverse set of application benchmarks (Table 4). These benchmarks have different characteristics of memory access patterns and page table walks (Figure 4), allowing us to comprehensively analyze the performance of DMT and related translation designs.

6 Evaluation

In the evaluation, we measure the performance of DMT (including pvDMT), in terms of (1) page table walk latency and (2) application execution time. The measurement is done both with and without enabling Linux's Transparent Huge Page (THP). We use the vanilla Linux/KVM on the x86-64 architecture as our *baseline* system (§2) to understand the speedup DMT brings. We compare DMT and pvDMT with four recent translation designs as our references: ECPT [64], FPT [59], Agile Paging [20], and ASAP [45].

We also measure the overhead of DMT incurred by OS work for managing VMA-to-TEA mapping, TEAs, as well as extra memory due to the eager allocation of TEAs (§6.3).

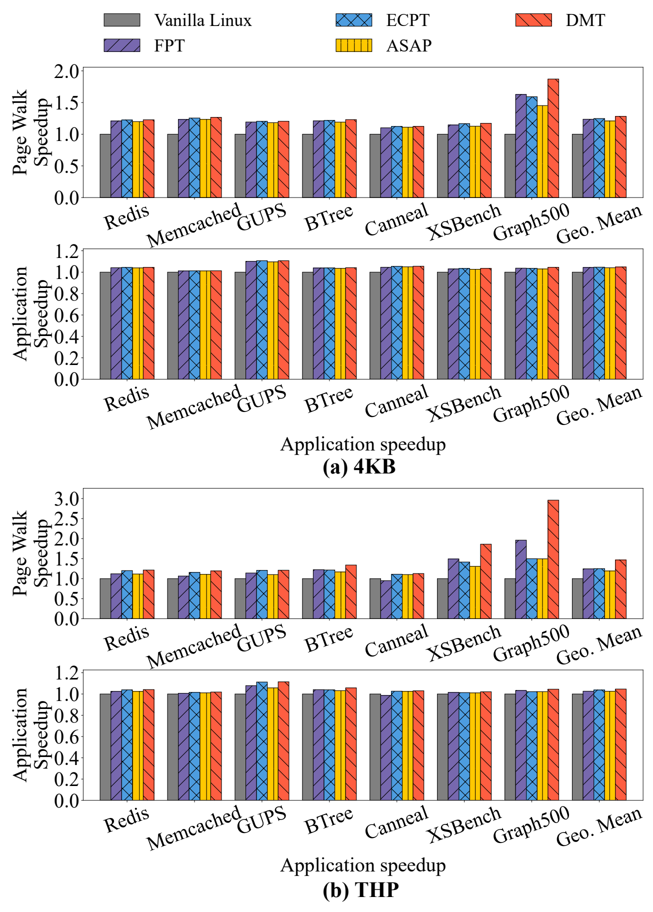

Figure 14. Speedup of page table walk and application execution in (a) 4KB and (b) THP of DMT and other advanced translation designs over baseline (Linux on x86) in a native environment.

6.1 DMT Performance over Baseline

DMT significantly outperforms the baseline in native (§6.1.1), single-level virtualization (§6.1.2), and nested virtualization (§6.1.3) environments. The DMT registers cover 99+% translation requests in all three environments. We discuss how DMT compares to other advanced designs in §6.2.

6.1.1 Native. Figure 14 shows the speedups of page table walk and application execution time across the benchmarks in the evaluated systems in the native environment. On average, DMT offers a speedup of 1.28x in page table walk and 1.05x in application execution time over the baseline, if only 4KB pages are used (Figure 14a). With THP, DMT offers an average speedup of 1.46x in page table walk and 1.05x in application execution time (Figure 14b).

The performance improvement is attributed to DMT's saving of sequential fetching multiple higher-level PTEs. THP further enlarges the benefits of DMT, because it reduces the ratios of fetching the last-level PTEs from main memory

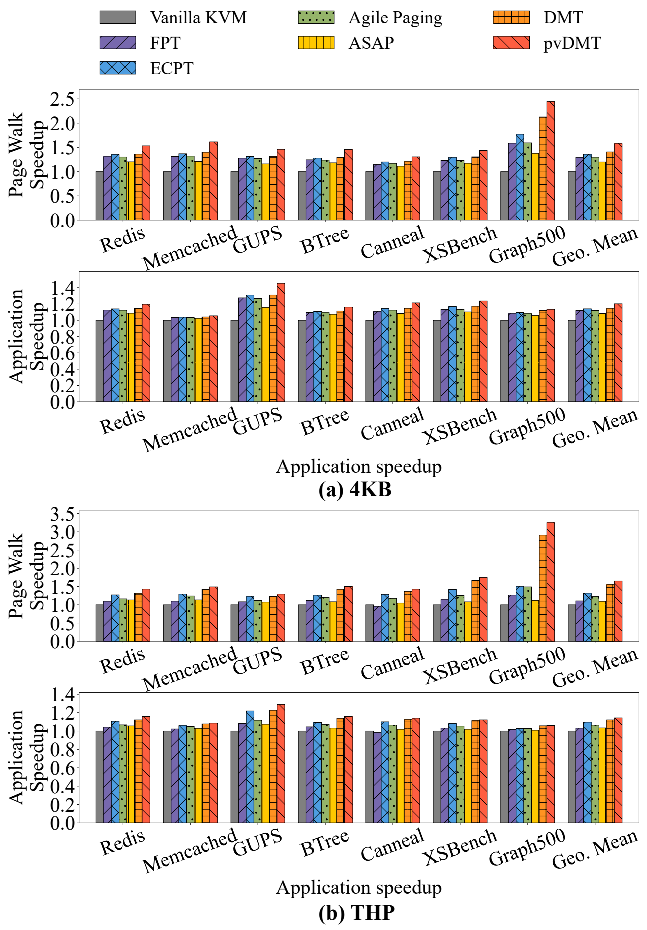

Figure 15. Speedup of page table walk and application execution in (a) 4KB and (b) THP of DMT and pvDMT and the other advanced designs over baseline (Linux/KVM on x86) in a virtualized environment.

and amplifies the effect of skipping page table walk. We will observe this behavior even more in the virtualized environments in §6.1.2 and §6.1.3.

6.1.2 Virtualization. DMT, especially pvDMT, leads to more substantial performance improvements in virtualized environments. Figure 15a shows that DMT exhibits a speedup of 1.41x in page table walk and 1.15x in application execution time if only 4KB pages are used; with pvDMT, the speedup of page table walk is 1.58x and the speedup of application execution time is 1.20x. With THP, DMT speeds up page table walk by 1.55x and application execution time by 1.12x (Figure 15b). With pvDMT, the speedup of page table walk is 1.65x and the speedup of application execution time is 1.14x.

Note that DMT/pvDMT with THP has larger speedups of a page table walk latency but lower in application execution time than without THP, because with THP page table walk overhead is much reduced in the baseline.

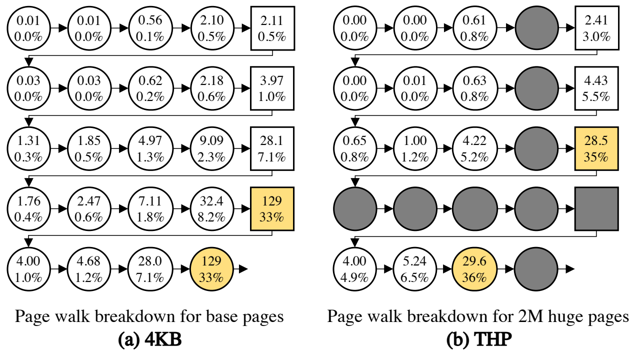

Figure 16. Breakdown of nested page table walks. The top number is the average cycles of accessing a PTE. The bottom number is the proportion of time spent on each PTE with regard to average page table walk latency. pvDMT fetches only two PTEs (highlighted).

Figure 16 breaks down the page table walk overhead of pvDMT in Redis (Table 4). For 4KB pages (Figure 16a), last-level PTEs are less cached. By directly fetching the last-level PTEs, pvDMT reduces the average page table walk latency to 66% (33% + 33%) of the baseline. With THP, last-level PTEs are better cached, hence lower cost to fetch (Figure 16b). As a result, pvDMT reduces the page table walk latency to 71% (35% + 36%) of the baseline. Other workloads show similar characteristics. For example, in Canneal, pvDMT reduces the page table walk latency to 76% (38% + 38%) and 69% (37% + 32%) of the baseline for 4KB pages and THP respectively. Compared to Redis, in Canneal, 2MB PTEs are more effectively cached (and thus cheaper to access); therefore, with THP, pvDMT's reduction of page table walk latency over the baseline is larger than 4KB pages. In summary, even in the cases where last-level PTEs are cached, DMT/pvDMT still brings substantial benefits.

Different workloads differ in memory access patterns. Despite for most workloads, last-level PTEs are rarely cached during page table walks, the caching patterns of higher-level PTEs are different. Since DMT eliminates the cost of fetching higher-level PTEs, the effect of the removal is different. This leads to the difference in speedups of page table walk latency. For workloads with access patterns that enable effective PTE caching (e.g., Graph500), DMT mostly saves accesses to the caches. Since the last-level PTEs are better cached, accesses to them cause low latency and lead to a higher speedup in page table walk latency for DMT. For workloads with access patterns that are hard to cache (e.g., Redis with a large working set), DMT saves accesses to main memory as well as CPU caches. In this case, though the speedup is lower for these workloads, the absolute saving is higher.

6.1.3 Nested Virtualization. DMT/pvDMT is the first hardware-assisted translation design for nested virtualization. We measure performance improvements of pvDMT over the baseline, Nested KVM.

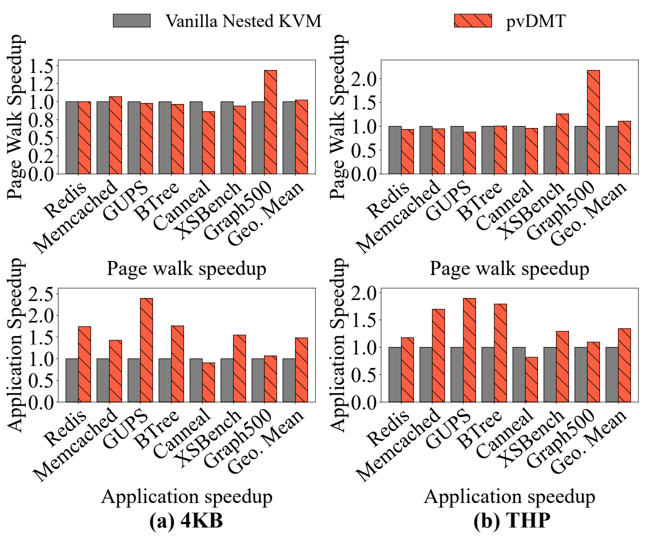

Figure 17. Speedup of page table walk and application execution in (a) 4KB and (b) THP of pvDMT over the baseline (Nested KVM).

Figure 17a shows that pvDMT speeds up application execution of the baseline by 1.48x on average if only 4KB pages are used. pvDMT's page table walk is only slightly faster (1.02x on average) than the baseline because the baseline uses shadow paging to avoid three-dimensional page table walks. As last-level PTEs are harder to cache, pvDMT takes three sequential memory accesses in most of the cases. Meanwhile, since pvDMT eliminates the shadow paging overhead caused by VM exits (§2.1.3), it significantly improves end-to-end performance.

With THP, pvDMT greatly outperforms the baseline even in terms of page table walk overhead, despite its need to translate one more level than shadow paging, because the PTEs are effectively cached. As shown in Figure 17b, with THP, pvDMT speeds up page table walk by 1.11x and application execution by 1.34x on average.

6.2 Comparison with Other Advanced Designs

Figures 14 and 15 show detailed comparisons of DMT/pvDMT with the other advanced designs (ECPT [64], FPT [59], Agile Paging [20], and ASAP [45]) in the native and virtualized environments. Table 5 summarizes the results, showing that DMT/pvDMT outperforms the state of the arts.

6.2.1 Translation Designs. Table 6 lists the number of *sequential* memory references of these different designs. We comparatively discuss each design with DMT/pvDMT, with a focus on virtualized environments.

ECPT [64, 66] also directly fetches PTEs using cuckoo hashing. We did not expect DMT (without pvDMT) to outperform ECPT in native or virtualized environments, because they take the same number of sequential memory accesses. Instead, DMT can be viewed as a design to achieve ECPT

Table 5. DMT/pvDMT's speedups of a page table walk over other advanced designs (§6.2). pvDMT is used for comparisons in Virtualized (4K/THP). The values are the geometric means.

Environment	FPT	ECPT	Agile Paging	ASAP
Native (4KB)	1.04x	1.03x	N/A	1.06x
Native (THP)	1.18x	1.17x	N/A	1.23x
Virtualized (4KB)	1.22x	1.16x	1.21x	1.31x
Virtualized (THP)	1.49x	1.25x	1.34x	1.51x

Table 6. The number of sequential memory accesses of other translation designs in different environments.

Design	Native	Virtualization	Nested Virt.
pvDMT	**1**	**2**	**3**
ECPT	1	3	N/A
FPT	2	8	N/A
Agile Paging	N/A	4–24	N/A
ASAP	4	24	N/A

performance with x86 compatibility. Interestingly, our evaluation shows that DMT, even without pvDMT, brings non-trivial improvements over ECPT in both native and virtualized environments. The reasons are: (1) compared with ECPT, DMT does not spend cycles on hash calculations and cuckoo walk cache lookup, and (2) DMT issues much fewer parallel lookups compared with ECPT. pvDMT further reduces the number of sequential memory references from three to two. On average, pvDMT outperforms Nested ECPT [66] in terms of page table walk latency by 1.16x (1.25x with THP).

FPT [59] flattens page tables by merging adjacent page table levels. Specifically, it merges L_4 with L_3 and L_2 with L_1, reducing memory references from four to two in a native environment. In a virtualized environment, FPT requires eight sequential memory references for each two-dimensional page table walk. pvDMT outperforms FPT in terms of page table walk latency by 1.22x (1.49x with THP) on average.

Agile Paging [20] combines nested paging and shadow paging to accelerate translation in virtualized systems. It starts with the shadow page table for the higher levels of the radix page table and switches to nested paging for the lower levels of the page table. Compared to pvDMT, Agile Paging requires many more memory references for a translation. On average, pvDMT outperforms Agile Paging in terms of page table walk latency by 1.21x (1.34x with THP).

6.2.2 PTE Prefetching. DMT corroborates ASAP [45] in terms of the hardware extension and PTE management by the OS. DMT shows that optimizing the translation leads to significant benefits even when PTEs are prefetched into the caches, because sequentially fetching them one by one is still expensive, especially in virtualized environments.

In a virtualized environment, pvDMT outperforms ASAP by 1.31x (1.51x with THP) in terms of page table walk latency on average. Despite L_1 and L_2 entries being prefetched in

ASAP, a translation still takes a two-dimensional walk to sequentially fetch all the 24 PTEs. Moreover, the nature of nested page table walk inevitably creates a dependency chain of PTEs that cannot be prefetched in parallel (e.g., a host page table walk is required to resolve the hPA of a gPTE). Hence, if both host and guest PTEs are not cached, ASAP needs to fetch PTEs sequentially, taking hundreds of cycles. Lastly, ASAP may also pollute CPU caches.

6.3 DMT Overhead

DMT's runtime overhead mainly comes from OS work for managing VMA-to-TEA mappings (§4.2) and TEAs (§4.3). The management operations are mostly triggered by VMA creation and updates. In evaluated workloads, dynamic VMA changes are rare and VMA creation is mostly done at application initialization time. DMT makes the right tradeoff to optimize frequent memory address translation with infrequent overheads of VMA operations.

To quantify DMT's runtime overhead on real machines, we run DMT-Linux without hardware support. Specifically, we create a highly fragmented memory (using a fragmentation tool in [40] with a free memory fragmentation index of 0.99). Fragmentation increases DMT overhead. We record the execution time of all the management procedures during workload execution. Redis shows the largest overhead and it only adds on average 12ms, 120ms, and 598ms to application execution time in native, virtualized, and nested virtualized environments. Such overheads are negligible with regard to the application execution time (thousands of seconds).

We also measure hypercall overhead in pvDMT, using a microbenchmark that requests different sizes of TEAs. The TEA allocation time is 13.27 ms, 23.73 ms, and 48.07 ms in a virtualized environment and 15.67 ms, 24.55 ms, and 54.87 ms in nested virtualization, for 50 MB, 100 MB and 200 MB TEAs, respectively. The hypercall overhead (excluding memory allocation) is 1.88 μs in a virtualized environment and 10.75 μs in nested virtualization, incurred by the context switch (VM exit) and running KVM hypercall handling code.

DMT consumes extra memory space as it eagerly allocates memory space for TEAs. But, the extra memory space is negligible (<2.5% of the baseline). Specifically, in our experiments, DMT and Linux/KVM consume 247.2 MB and 241.3 MB of space to store page tables on average, respectively. Note that VMAs are created for *in-use* virtual memory regions, and most of them will be accessed.

DMT's hardware extension requires additional power and on-chip space. We use CACTI [8] with 22 nm technology to estimate its hardware cost. DMT consumes an additional 4.87 mW of leakage power and adds 0.03 mm^2 extra on-chip space per MMU. The overheads are marginal based on the fact that the Thermal Design Power of the modeled CPU, Intel Xeon Gold 6138, is 125 W [31] and its die size is 694 mm^2 [78] with 14 nm technology.

7 Discussion and Limitations

Assumptions. DMT assumes that a handful of VMAs or VMA clusters with small bubbles can effectively cover the working set of data-intensive applications. We validate this assumption in §2.3 using various applications and benchmarks. New memory allocators like TCMalloc [21] would further bring down the number of VMAs by coalescing. However, applications with highly fragmented VMAs may not benefit greatly from DMT, if a few registers cannot encode the working set effectively. In such cases, more translations will fall back to the original x86 page table walks. DMT also assumes that VMAs are updated infrequently after creation. DMT targets data-intensive workloads. Such workloads do not have frequent VMA updates—they typically allocate memory at the initialization time [10, 25, 45]. Certainly, it is possible that specific workloads change VMAs frequently; such workloads may not benefit from DMT.

Paravirtualization. The use of paravirtualization (i.e., pvDMT) effectively optimizes DMT for virtualized environments. However, it has several known shortcomings. Besides the hypercalls overhead, paravirtualization requires both the guest and the host to support DMT-Linux; otherwise, DMT falls back to the original x86 radix-based translation. If the hardware extension is adopted and available, we hope that DMT-Linux can be upstreamed to Linux/KVM.

Contiguity requirement of TEA. TEAs need contiguous physical memory space. Despite TEAs being orders of magnitude smaller than VMAs (e.g., a 200MB TEA is needed for 100GB data with 4KB pages), memory contiguity could be scarce in long-running servers [79]. Recent work [85] shows that by isolating unmovable regions, 1GB contiguity can be commonly allocated. Also, new contiguity-aware allocation and de-fragmentation mechanisms are developed [77]. We expect the megabyte-level contiguity of TEA to be acceptable. DMT splits TEAs when contiguity cannot be found.

Eager TEA allocation. DMT eagerly allocates physical memory for TEAs and might waste space compared to the lazy allocation of x86. Our evaluation shows that the extra memory space is negligible (§6.3). However, there exist workloads where eager allocation can be more wasteful, e.g., `mmap`ping a 1TB file to memory but accessing a small portion of it. For such workloads, more advanced TEA allocation policies can be employed, e.g., on-demand allocation of small-sized TEAs with dynamic expansion. As TEA allocation is done by the OS, the policy can be decided as per workloads.

8 Related Work

Accelerating address translation. DMT is inspired by prior work on redesigning memory translation to reduce its overhead [3–5, 10, 12, 20, 23, 26, 54, 59, 64, 66–68, 70, 75, 80]. We compare the performance of DMT with the recent designs in §6.2 and show that DMT/pvDMT can further reduce memory translation overhead. At a high level, DMT differs from prior work with two main design principles: (1) co-designing memory translation hardware with OS memory management, which greatly simplifies the design and is transparent to user applications, and (2) avoiding disruptive changes to the existing architecture and using backward compatible designs to achieve minimized overhead.

Huge page. Using huge pages to improve TLB coverage and shorten page table walks is practical and often only involves software changes [29, 35, 36, 40, 43, 53, 55–57]. However, huge pages cannot fundamentally address the translation overhead as it is less scalable and cannot eliminate all the sequential page table walks. DMT is complementary to and can benefit from huge pages, as shown in our evaluation.

Translation caching and prefetching. Prior work reduces translation overhead by using PTE prefetching [45] and caching [9, 25, 63]. As we discuss in §6.2.2, prefetching and caching can further unleash the power of DMT by bringing PTEs into the caches in advance. Specifically, some building blocks of DMT are compatible with PTE prefetchers, specifically ASAP [45]. We believe that a unified infrastructure to support both prefetching and DMT is viable.

TLB efficiency. Prior work has greatly improved TLB efficiency [6, 10, 19, 33, 34, 38, 58, 60, 61]. However, as TLB capacity does not increase at the rate as memory capacity and irregular memory access patterns of emerging workloads, off-TLB translation efficiency is increasingly relevant.

9 Concluding Remarks

We set to explore a practical address translation design for minimizing memory translation overhead in virtualized cloud environments. We settled with Direct Memory Translation (DMT), a hardware-software extension that directly fetches the last-level page table entries containing the translation while maintaining backward compatibility with x86-based virtual memory. DMT makes us believe that an incremental path towards highly efficient virtual memory is viable, even for virtualized environments including nested virtualization.

Acknowledgement

We thank our shepherd, Idan Yaniv, and the anonymous reviewers for their insightful comments that greatly improve the paper. We thank Jovan Stojkovic and Josep Torrellas for valuable discussions. We thank Neil Zhao for the discussion on side channels in paravirtualization systems. This work was funded by NSF CNS-1956007 and an Intel TSA grant.

References

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*.

[2] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. 2010. The Evolution of an x86 Virtual Machine Monitor. *ACM SIGOPS Operating Systems Review* (Dec. 2010), 3–18.

[3] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*.

[4] Sam Ainsworth and Timothy M Jones. 2021. Compendia: Reducing Virtual-Memory Costs via Selective Densification. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM'21)*.

[5] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[6] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[7] Microsoft Azure. 2017. Nested Virtualization in Azure. https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/.

[8] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (June 2017), 1–25.

[9] Thomas W Barr, Alan L Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*.

[10] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

[11] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.

[12] Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Rob Johnson, Sudarsun Kannan, William Kuszmaul, Nirjhar Mukherjee, Don Porter, Guido Tagliavini, Janet Vorobyeva, and Evan West. 2021. Paging and the Address-Translation Problem. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'21)*.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08)*.

[14] Google Cloud. 2023. About nested virtualization. https://cloud.google.com/compute/docs/instances/nested-virtualization/overview.

[15] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[16] Advanced Micro Devices. 2023. AMD64 Architecture Programmer's Manual: Volumes 1-5. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf.

[17] Yu Du, Miao Zhou, Bruce R. Childers, Daniel Mossé, and Rami Melhem. 2015. Supporting Superpages in Non-Contiguous Physical Memory. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA-21)*.

[18] DynamoRIO. 2023. Dynamic Instrumentation Tool Platform. https://github.com/DynamoRIO/dynamorio.

[19] Jayneel Gandhi, Arkaprava Basu, Mark D Hill, and Michael M Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*.

[20] Jayneel Gandhi, Mark D Hill, and Michael M Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA'16)*.

[21] Google. 2023. TCMalloc. https://github.com/google/tcmalloc.

[22] Mel Gorman. 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall.

[23] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim Nael Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. 2023. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*.

[24] Graph 500 Steering Committee. 2017. Graph 500 Benchmark Specification. https://graph500.org/?page_id=12.

[25] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. 2021. Rebooting Virtual Memory with Midgard. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA'21)*.

[26] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*.

[27] HPC Challenge Benchmark. 2022. RandomAccess: GUPS (Giga Updates Per Second). https://hpcchallenge.org/projectsfiles/hpcc/RandomAccess.html.

[28] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. 2023. PVM: Efficient Shadow Paging for Deploying Secure Containers in Cloud-native Environment. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*.

[29] AH Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[30] Intel. 2017. 5-Level Paging and 5-Level EPT White Paper. https://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html.

[31] Intel. 2017. Intel Xeon Gold 6138 Processor 27.5M Cache 2.00 GHz Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/120476/intel-xeon-gold-6138-processor-27-5m-cache-2-00-ghz.html.

[32] Intel. 2023. Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3C. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[33] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: High-Performance Address Translation by Extending TLB Reach of GPU-Accelerated Systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (March 2019), 1–24.

[34] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. 2023. Making Dynamic Page Coalescing Effective on Virtualized Clouds. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*.

[35] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, Yiming Du, Xiaoning Ding, and Tianyin Xu. 2023. HugeGPT: Storing Guest Page Tables on Host Huge Pages to Accelerate Address Translation. In *Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT'23)*.

[36] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*.

[37] Konstantinos Kanellopoulos, Rahul Bera, Kosta Stojiljkovic, F. Nisa Bostanci, Can Firtina, Rachata Ausavarungnirun, Rakesh Kumar, Nastaran Hajinazar, Mohammad Sadrosadati, Nandita Vijaykumar, and Onur Mutlu. 2023. Utopia: Efficient Address Translation using Hybrid Virtual-to-Physical Address Mapping. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-56)*.

[38] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[39] Osang Kwon, Yongho Lee, and Seokin Hong. 2022. Pinning Page Structure Entries to Last-Level Cache for Fast Address Translation. *IEEE Access* 10 (Oct. 2022), 114552–114565.

[40] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[41] Henry M. Levy and Peter H. Lipman. 1982. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer* 15, 3 (March 1982), 35–41.

[42] Jin Tack Lim and Jason Nieh. 2020. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[43] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S McKinley, and Paul Turner. 2021. Adaptive Huge-Page Subrelease for Non-moving Memory Allocators in Warehouse-Scale Computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM'21)*.

[44] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. 2017. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*.

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

[46] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. 2021. 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 (ASPLOS'21)*.

[47] Chandrashis Mazumdar, Prachatos Mitra, and Arkaprava Basu. 2021. Dead Page and Dead Block Predictors: Cleaning TLBs and Caches Together. In *Proceedings of the 27th IEEE International Symposium on High Performance Computer Architecture (HPCA-27)*.

[48] Memcached. 2023. memcached - a distributed memory object caching system. https://memcached.org.

[49] Microsoft. 2023. Frequently Asked Questions about Windows Subsystem for Linux. https://learn.microsoft.com/en-us/windows/wsl/faq.

[50] Microsoft. 2023. Virtualization-based Security (VBS). https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs.

[51] Mitosis Project. 2020. Mitosis Workload Btree. https://github.com/mitosis-project/mitosis-workload-btree.

[52] Ingo Molnar. 2009. Performance Counters for Linux. https://lwn.net/Articles/337493/.

[53] Juan Navarro, Sitaram Iyer, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*.

[54] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (May 2004), 122–144.

[55] Ashish Panwar, Sorav Bansal, and K Gopinath. 2019. 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 (ASPLOS'19)*.

[56] Ashish Panwar, Aravinda Prasad, and K Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*.

[57] Chang Hyun Park, Sanghoon Cha, Bokyeong Kim, Youngjin Kwon, David Black-Schaffer, and Jaehyuk Huh. 2020. Perforated Page: Supporting Fragmented Memory Allocation for Large Pages. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[58] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[59] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. 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 (ASPLOS'22)*.

[60] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. 2014. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA-20)*.

[61] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

[62] Redis 2023. Redis. http://redis.io/.

[63] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[64] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[65] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. 2023. Memory-Efficient Hashed Page Tables. In *Proceedings of the 29th IEEE International Symposium on High-Performance Computer Architecture (HPCA-29)*.

[66] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2022. 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'22)*.

[67] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. 2020. CARAT: A Case for Virtual Memory through Compiler- and Runtime-Based Address Translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*.

[68] Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, Simone Campanoni, and Peter Dinda. 2022. CARAT CAKE: Replacing Paging via Compiler/Kernel Cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*.

[69] Madhusudhan Talluri, Mark D. Hill, and Yousef A. Khalidi. 1995. A New Page Table for 64-bit Address Space. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*.

[70] Boris Teabe, Peterson Yuhala, Alain Tchana, Fabien Hermenier, Daniel Hagimont, and Gilles Muller. 2021. (No)Compromis: Paging Virtualization Is Not a Fatality. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'21)*.

[71] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *Proceedings of the International Conference on Physics of Reactors (PHYSOR 2014)*.

[72] VMware. 2009. Performance Evaluation of Intel EPT Hardware Assist. https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[73] VMware. 2023. Support for running ESXi as a nested virtualization solution. https://kb.vmware.com/s/article/2009916.

[74] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*.

[75] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. 2011. Selective Hardware/Software Memory Virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'11)*.

[76] WikiChip. 2019. Intel Xeon Gold 6138. https://en.wikichip.org/wiki/intel/xeon_gold/6138.

[77] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. 2023. LLFree: Scalable and Optionally-Persistent Page-Frame Allocation. In *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC'23)*.

[78] x86 cpus' Guide. 2023. Intel Xeon Gold 6138. https://www.x86-guide.net/en/cpu/Intel-Xeon-Gold-6138-cpu-no6271.html.

[79] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[80] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'16)*.

[81] Huaisheng Ye. 2021. Introduction to 5-Level Paging in 3rd Gen Intel Xeon Scalable Processors with Linux. https://lenovopress.lenovo.com/lp1468.pdf.

[82] Martin Yip. 2018. Running Hyper-V on Amazon EC2 Bare Metal Instances. https://aws.amazon.com/blogs/compute/running-hyper-v-on-amazon-ec2-bare-metal-instances/.

[83] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*.

[84] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*.

[85] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. 2023. Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA'23)*.