Do-It-Yourself Virtual Memory Translation

Hanna Alam[1] Tianhao Zhang[2] Mattan Erez[2] Yoav Etsion[1]
Technion - Israel Institute of Technology[1] The University of Texas at Austin[2]
{ahanna,yetsion}@tce.technion.ac.il
{thzheng,mattan.erez}@utexas.edu

ABSTRACT

In this paper, we introduce the Do-It-Yourself virtual memory translation (DVMT) architecture as a flexible complement for current hardware-fixed translation flows. DVMT decouples the virtual-to-physical mapping process from the access permissions, giving applications freedom in choosing mapping schemes, while maintaining security within the operating system. Furthermore, DVMT is designed to support virtualized environments, as a means to collapse the costly, hardware-assisted two-dimensional translations. We describe the architecture in detail and demonstrate its effectiveness by evaluating several different DVMT schemes on a range of virtualized applications with a model based on measurements from a commercial system. We show that different DVMT configurations preserve the native performance, while achieving speedups of 1.2× to 2.0× in virtualized environments.

CCS CONCEPTS

• **Software and its engineering** → **Memory management**; **Virtual memory**; **Virtual machines**; **Operating systems**;

KEYWORDS

TLB, virtual memory, virtual machines, address translation

ACM Reference format:
Hanna Alam[1] Tianhao Zhang[2] Mattan Erez[2] Yoav Etsion[1] Technion - Israel Institute of Technology[1] The University of Texas at Austin[2] . 2017. Do-It-Yourself Virtual Memory Translation. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 12 pages.
https://doi.org/10.1145/3079856.3080209

1 INTRODUCTION

Page-based virtual memory decouples an application's view of its memory resources from their physical layout. This decoupling enables operating systems (OS) and hypervisors to eliminate memory fragmentation and improve performance and memory utilization by managing physical memory resources while providing applications with a view of a contiguous address space that is isolated from all other applications.

Processor architectures (e.g., Intel [1], AMD [4], ARM [6]) typically employ radix tree page tables to dynamically map virtual memory addresses to physical ones, which enables them to support sparsely populated virtual address spaces. The OS manages physical resources at fixed, page-sized allocation units with address translation determined by the OS and stored in the radix-tree page table. Mapping a virtual address to a physical one thus requires traversing the radix tree, or "walking the page table", which incurs multiple memory accesses. On Intel's x86-64 platforms, for example, the page table's radix is four such that a single load operation may require up to five memory accesses for translation and data fetches. To mitigate this overhead, modern processors make aggressive use of multi-level translation look-aside buffers (TLBs) to cache memory mappings. Furthermore, partial walk caches are used to cache partial traversals through the radix tree [7, 12].

Another common approach to reducing translation overheads is to increase the page size. This has the dual effect of increasing the TLB reach and reducing the radix tree depth. However, the coarser granularity of large pages can curtail lightweight and agile system memory management [34]. Furthermore, even with coarser pages, the page table structure is still tuned for sparsely populated virtual address spaces, which is at odds with some modern workloads that pre-allocate large contiguous regions of virtual memory. Examples of such workloads include data processing, scientific applications, and virtual machines (VMs). These workloads, therefore, have large regions of fully populated virtual address spaces that can benefit from simpler mapping structures.

The mapping overhead of page traversal is most evident in VM workloads, in which a hypervisor allocates a large virtual memory region to emulate the guest VM's physical memory. The guest OS then manages its own mapping of *guest virtual addresses* (gVA) to *guest physical addresses* (gPA). This nested memory mapping [11], supported by commercial processors [1, 4, 6], squares the number of accesses needed to map a guest virtual address to its *host physical address* (hPA). On Intel and AMD architectures, for example, the gVA-to-hPA mapping may take up to 24 memory accesses.

In this paper, we present the *Do-It-Yourself Virtual Memory Translation* (DVMT) architecture, a virtual memory architecture that enables applications and virtual machine monitors (VMMs) to customize virtual-to-physical memory mappings. DVMT decouples address mapping from access permission validation. This decoupling enables the OS/hypervisor to delegate the virtual-to-physical mapping to the application, while still enforcing its own security and isolation policies. Furthermore, DVMT maintains the concept of memory paging and swapping, which enables the OS/hypervisor to incrementally reclaim physical memory pages without terminating any application.

DVMT is intended to complement the existing virtual memory architecture and enable applications to optimize the mapping process only for those specific memory regions that can benefit from a

ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080209

customized virtual-to-physical mapping. We envision that most customized mappings will be handled by specialized memory allocators or tuned operating systems (library OSs, VM images, or containers).

The DVMT architecture operations can be summarized as follows:

Virtual-to-physical mapping and translation. Using DVMT, an application can request physical frames from the OS/hypervisor with address properties that allow the application to construct its customized *do-it-yourself* (DIY) mapping. For example, an application may request a large contiguous allocation that can then be directly mapped or even make multiple requests and construct a specialized acceleration structure, such as a hashed page table. The application then registers its own mapping function as a helper thread with the OS/hypervisor. On a TLB miss from the DVMT virtual address range, the helper thread is triggered to compute the mapping and insert it into the TLB. Other TLB misses use the system's default page walk. Notably, an application's customized (DIY) translation scheme is private to the application and is independent of other applications, which may each use their own customized mappings.

Enforcing memory protection. The validity of customized memory mappings is enforced using a minimal permissions mechanism, inspired by capability systems [29], which is invoked on each TLB insertion. The OS/hypervisor maintains a system-global array that tags each physical frame with an ID and associates the ID and access permissions with applications; we refer to each $\langle ID, permissions\rangle$ tuple as a *token*. To validate a DVMT mapping, the processor checks that the TLB-filling thread holds a token that is associated with the physical frame. We describe a simple token table that requires at most a single memory access for the protection check. The tokens are granted only by the OS/hypervisor and therefore maintain the same permissions model as current systems.

The proposed DVMT design thus enables VM workloads to map gVA to hPA with just a single memory access even when all translation caches miss: a direct segment translation is done with no memory accesses and the protection capability check requires a single memory access. This is contrast to the 24 accesses required for gVA-to-hPA mapping on current the x86-64 architecture.

To summarize the contributions of this paper:

- We introduce the DIY virtual memory translation (DVMT) architecture, which enables runtime systems and applications to employ customized virtual-to-physical mapping using helper threads.
- We propose a new light-weight capability system that validates DIY memory mappings and enforces memory protection using a single memory access.
- We evaluate DVMT using a set of scientific, data processing, and VM workloads together with several customized translation schemes consisting of a hash table [41] , a flat indirection array, and a direct segment [10]. Our evaluation shows that using DVMT in the hypervisor can improve the performance of VM workloads, achieving an average speedup of 1.2–1.5× for the different schemes. We further show that different configurations of DVMT schemes in both hypervisor and guest achieve average speedups of 1.3–2.0×.

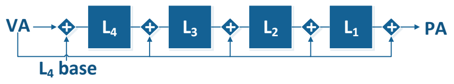


Figure 1: x86-64 native VA→PA Translation

2 ON THE NEED FOR CUSTOM MEMORY TRANSLATION: AN X86-64 CASE STUDY

Prevalent computer architectures (e.g., Intel, ARM) employ radix trees to translate virtual addresses (VA) to physical addresses (PA) and to enforce the memory access protection dictated by the operating system. The key benefit of using a radix tree is that it can serve both sparse and dense virtual address spaces, but this generality comes at a cost. Poor memory access locality workloads, which commonly characterize modern virtual machines and large-memory databases, can benefit from more efficient mapping methods.

In this section we examine common radix tree-based virtual memory mechanisms, using the x86-64 implementation as a case study, and discuss the performance implications of this design choice.

Readers who are familiar with the overheads of virtual memory translation can skip directly to Section 2.2, which argues for customizing memory translations.

2.1 Memory translation overheads in X86-64

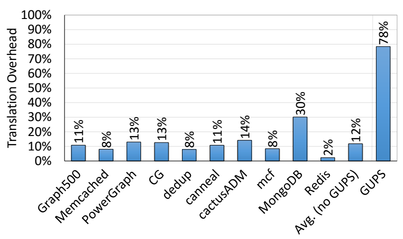
2.1.1 Native virtual address translation.

x86-64 processors use a radix tree of depth 4 for virtual-to-physical translation in native application. We denote each layer in the tree as L_i, where $1 \leq i \leq 4$ (with L_4 being the root of the tree). Each node in the tree consists of a table representing the nodes in the subsequent level of the tree. The node populates a single physical memory frame. Each table entry maintains the physical frame number (pointer) of the child node and the OS imposed permissions for the entire subtree. Leaf nodes hold the target frame number to which the source virtual page is mapped and its permissions.

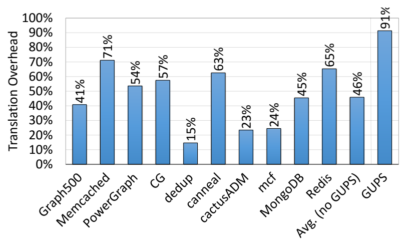
The translation process, often referred to as a *page table walk*, is depicted in Figure 1. The root of the tree is pointed to by the architectural page table register (*CR3* in x86-64) in the form of the physical frame that stores the root node L_4. The translation process is iterative and traverses a path from the root to a leaf node. In each step, subsequent bit sets from the source virtual address are used to index the tables in subsequent nodes along the path. For example, in a 4KB page size configuration, bits 39–47 in the VA (current x86-64 implementations support 48-bit virtual address spaces) are used to index the table in the L_4 root node. The indexed entry provides the frame number of the L_3 node that is next along the search path.

The recursive structure of the tree provides support for sparse virtual address spaces without incurring excessive storage overheads. Any table entry in any node in the tree can be marked as invalid, thereby indicating that the entire virtual address space represented by the subtree is not mapped. When the entire subtree is not mapped, nodes in the tree need not be allocated, thereby saving memory.

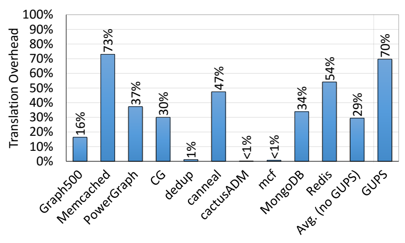
Discussion. The performance penalty of a valid page walk is four memory accesses. This excessive overhead is mitigated using aggressive translation caches. The translation lookaside buffer (TLB) is the primary cache holding recent virtual-to-physical mappings.



(a) Native applications, 4KB pages.



(b) Applications running in virtual machine, g4KB/h4KB pages.



(c) Applications running in virtual machine, g2MB/h2MB pages.

Figure 2: Fraction of total run time devoted to page table walks on Haswell for different machine and application configurations (see Section 5 for a description of benchmarks and methodology).

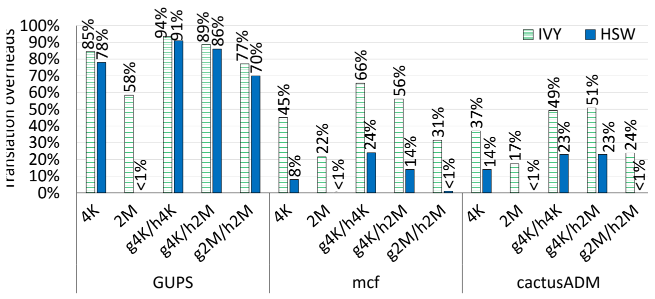


Figure 3: The translation overheads on Haswell vs. Ivy Bridge processors (g4K/h2M indicates using 4K pages in the guest and 2M in the host).

The TLB itself, however, must be small (up to 64 entries on the Intel Haswell (HSW) processor [1]) since it resides on the memory access's critical path: as data and instruction caches are physically tagged, the translation must be performed before the cache access. To palliate the small size of the TLB, contemporary processors employ a 2nd level TLB (STLB), partial walk caches (PWCs), and caching of page tables in data caches. PWCs reduce the number of memory accesses required to perform a virtual-to-physical translation by caching virtual addresses to the physical address of one intermediate tree node along the translation path.

Aggressive caching accelerates translation when workloads exhibit locality in virtual address space. For example, because Intel has increased the translation caches in the Haswell processor, we see that native translation overheads are minor, as shown in Figure 2a (the benchmarks and methodology are described in Section 5).

Figure 3 illustrates the performance impact of Intel's modifications to the translation caches between Ivy Bridge and Haswell processors (most notably, enlarging the PWCs and STLB). The figure presents the translation overheads observed on both architectures for select benchmarks, namely GUPS, mcf, and cactusADM.

We see that GUPS's performance is hardly improved in Haswell, as this benchmark generates a random memory access trace. In contrast, benchmarks that exhibit non-random memory access patterns (e.g., mcf and cactusADM) experience a major reduction of the translation overhead when running natively. Moreover, STLB caching of 2MB pages in Haswell effectively eliminates the translation overheads for virtualized environments that use large 2MB pages in both guest and host. Nevertheless, Haswell still experiences substantial translation overheads when using small 4KB pages in virtualized environments. Since 4KB pages are crucial for memory usage optimizations, Haswell's performance motivates the design of new translation mechanisms.

2.1.2 Translating addresses in virtual machines.

Virtual machines (VMs) require additional complexity when dealing with virtual memory translation. Each running process in a virtual machine needs its own isolated virtual address space. Consequently, the system must provide translation from guest virtual addresses (gVA) to a physical address on the host machine (hPA). This translation is performed using nested page tables. In each VM, the guest OS sets up a page table to map the gVA to a guest "physical address" (gPA). Since the host emulates the guest physical address space using a region in the host physical address space, a gPA is effectively a host virtual address (hVA). A hVA is translated to a hPA using per VM page tables in the hypervisor.

The main problem with the nested gVA-to-gPA translation is that it dramatically increases the number of memory accesses required by each of the nested radix tree page tables, as depicted in Figure 4, incurring up to 24 memory accesses for a single walk. This is because each node in the guest radix tree points to a node in the next level using a guest physical frame number, and thus each transition between levels in the guest radix tree gL_{i+1} to gL_i incurs a full gPA-to-hPA using the VM's page table in the hypervisor. This is appropriately referred to as a two-dimensional (2D) page walk. Notably, modern architectures execute 2D page walks directly in hardware using the conventional page table walker with modifications to the traversing algorithm.

Discussion. The dramatic increase in the number of memory accesses incurred by each page walk affects the run time of VM workloads. Figure 2b shows the fraction of the run time devoted to page table walks when running each of our benchmarks in a virtual machine, using 4KB pages in both the guest OS and the hypervisor. The figure shows that the page table walk overhead is a major contributor to the run time of VM workloads, standing at 46% on average (and 91% on GUPS).

As noted above, modern processors employ two main mechanisms to mitigate this overhead — large, multi-level TLBs and PWCs on the one hand, and support for large pages on the other. The effectiveness of the TLBs and PWCs is evident when examining translation

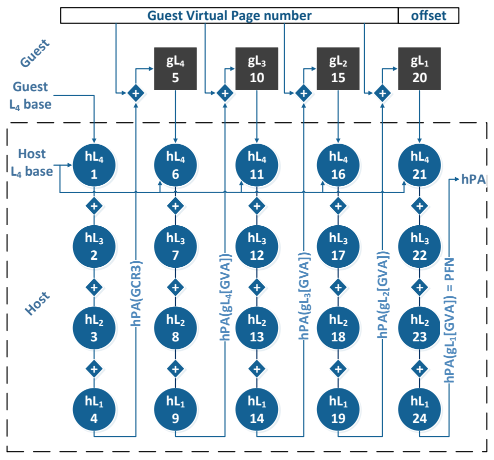


Figure 4: Two-dimensional gVA → hPA translation on the x86-64 architecture.

overheads in the native execution with 4KB pages (Figure 2a) vs. the g4KB/h4KB virtual machine configuration (Figure 2b). Notably, the overhead of address translation in VM workloads is less pronounced than we would expect: although each translation memory reference grows from 4 in native execution to 24 in virtualized execution (a factor of 6×), the runtime overhead only grows by a factor of 3.83 (from 12% to 46%).

The increased reliance on address translation caching suggests that a generic address translation mechanism, and specifically radix tree page tables, is too slow in the common case. We argue that the prevailing reliance on address translation caching is not a scalable solution. As modern workloads (e.g., data analytics, virtual machines) stress the memory system even further due to their dramatically increased memory footprint, the size and complexity of address translation mechanisms is only expected to grow. Indeed, vendors are already forced to continuously devote more resources to caching address translations, and modern processors include complex caching schemes such as extended PWCs (ePWC) support for virtualization and multi-level TLBs.

Large pages are commonly used to mitigate address translation overheads. Figure 2c shows the translation overheads when running the workloads in VMs, when both the guest OS and the hypervisor are configured to use 2MB pages. The figure shows that using large pages significantly reduces the translation overheads (down to 29% on average). Our results favor 2MB pages more than do previous studies, e.g., [10, 23, 34]; this is due to our use of an Intel Haswell processor, which introduces a new STLB for 2MB pages. The benefit of large pages comes from the increase in TLB and PWC span. Since 2MB pages are 512× larger than 4KB pages, each 2MB TLB cached translation is effectively equivalent to 512 translations of 4KB pages. In addition, using 2MB pages reduces the depth of the page table radix tree and thereby reduces memory accesses per TLB miss.

The use of large pages, however, greatly limits the hypervisor's flexibility and agility, making it difficult for it to efficiently monitor and manage the physical memory resources. One notable example, is the extensive use of data deduplication performed by modern hypervisors in order to reclaim physical memory. When the hypervisor notes that multiple physical frames store the same data, it simply augments the page tables pointing to those frames to share a single copy of the data and can thus reclaim the redundant memory frames. Yet the effectiveness of data deduplication drops dramatically as frame sizes increases [25, 34, 40].

2.2 The potential benefits of DIY virtual-to-physical translations

The analysis of the translation performance on the x86-64 architecture (Figure 2) teaches us that the generality of radix trees comes at a cost, and increasing the number of memory accesses incurred by address translations in VMs substantially affects system performance. Furthermore, the rigid memory management imposed by the use of large pages inhibits the agility required to manage memory in hypervisors. Consequently, we argue that efficient hardware support for application-specific memory address translation can leverage application semantics to reduce the number of memory accesses incurred by address translation.

Virtual machines are a striking example of applications that can benefit from such customization. VMs employ a large, contiguous virtual memory segment that is active throughout their execution. Yet the translation overheads they incur due to the traversal through generic radix trees are demonstrated by the difference in translation overheads depicted in Figure 2a, which examines native applications, and those depicted in Figure 2b, which examines the same applications in virtualized environments. This observation is also backed by previous studies that examined custom address translation for both native workloads (Basu *et al.* [10]) and virtual (Gandhi *et al.* [23]) environments.

We conclude that hardware support for application-specific address translation can go a long way towards mitigating translation overheads. Importantly, customization must not inhibit other effective optimization opportunities such as translation caching [7], and it must support memory paging with small pages, which facilitates efficient and agile OS/hypervisor memory management [34]. Notably, the aforementioned studies by Basu *et al.* [10] and Gandhi *et al.* [23] do not fulfill these requirements, as they rely on segmentation and the use of contiguous physical memory in the hypervisor (with a possibility to exclude physical frames with permanent hard faults using a bloom filter). The following sections describe our proposed extensions for existing address translation mechanisms that facilitate application-specific customizations.

3 RELATED WORK

DVMT facilitates generic, fast, workload-customized address translation by exposing physical memory resources to applications. It does so by uniquely incorporating 3 distinct hardware facilities: (1) a lightweight capability system [29], (2) hardware helper threads [17], and (3) software-managed TLBs [26]. In this section we survey past work aimed at mitigating address translation overheads and contrast it with the unique attributes that comprise DVMT.

Address translation overheads are commonly mitigated using large pages; this extends the coverage of the mappings cached in the TLBs [20, 21]. Large pages, however, greatly limit the hypervisor's ability to optimize resource utilization through extensive usage monitoring and memory deduplication [34]. Furthermore, supporting multiple page sizes increases hardware and OS complexity [37] and requires application support [21, 36]. TLB coverage can also be increased by clustering and coalescing page translations, as proposed by Pham *et al.* [33, 34] and Karakostas *et al.* [28]. Alternatively, Bhattacharjee *et al.* [13, 14], Lustig *et al.* [30], and Kandiraju and Sivasubramaniam [27] improved TLB efficiency either by sharing translations across cores using shared TLBs or through prefetching. These approaches are largely orthogonal to DVMT, which aims to reduce the translation time on TLB misses using fast, customized address translation.

Another method used to reduce translation overhead is hiding the translation latency by overlapping it with computation. Barr *et al.* [8] proposed SpecTLB, which predicts address translations. The predictions are validated while the processor speculatively executes the predicted memory accesses. Barr *et al.* [7] and Bhattacharjee [12] compared the partial caching of virtual and physical translations, and showed that virtually indexed translations deliver better performance. Zhang *et al.* [42] deferred address translation until a cache miss is encountered and data must be retrieved from physical memory. These studies are orthogonal to DVMT, as they aim to hide the overhead of existing translation mechanisms rather than minimize them.

The nesting of address translation in virtualized environments makes them most susceptible to the performance impact of address translation [3, 15]. Bhargava *et al.* [11] explored the 2D page traversal and proposed optimizations to partial walk caches. Ahn *et al.* [5] proposed to use a dedicated hardware translation unit for VMs that uses flat page tables, and thereby reduce the number of memory accesses required for gPA-to-hPA translation. A different approach, proposed by Wang *et al.* [38], dynamically decides whether to use nested or shadow page tables on the basis of application characteristics. Gandhi *et al.* [24] extend this approach with agile paging which allows for dynamic selection between both translation modes within the application's different memory regions. Unlike DVMT, these studies only target virtualized environments but ignore native applications. Moreover, they propose non-flexible translation schemes that may be inefficient for some workloads.

A number of studies proposed customizing address translation in the presence of software-managed TLBs, including the studies by Xiaotao *et al.* [16], Jacob and Mudge [26], and Engler *et al.* [19]. In contrast to DVMT, these systems suffer excessive translation overheads as they rely on frequent costly OS interventions and virtualization traps to initiate the translation and to enforce memory protection. Yaniv and Tsafrir [41] present a comparison between radix tree and hash table translations. Their study shows that carefully optimized hashed page tables may outperform existing PWC-aided x86-64 hardware and are inherently more scalable. DVMT can adopt these optimizations when implementing a custom, user-defined hash table translation scheme.

Shahar *et al.* [35] presented a case for a software address translation layer and paging system in GPUs, which enables the implementation of fully functional memory mapped files. DVMT similarly argues for bypassing the traditional translation scheme with an application-specific translation, while retaining the performance benefits provided by hardware translation caching (e.g., TLBs).

Finally, Gandhi *et al.* [23] proposed using segmentation in virtualized environments and big memory workloads, thereby replacing page walks with a much faster computation of segment offsets. However, this method inhibits resource utilization optimizations such as memory overcommit, paging, and swapping, which are common in virtualized environments [34]. DVMT, in contrast, supports (small-page) paging and facilitates optimizations such as memory overcommit and deduplication.

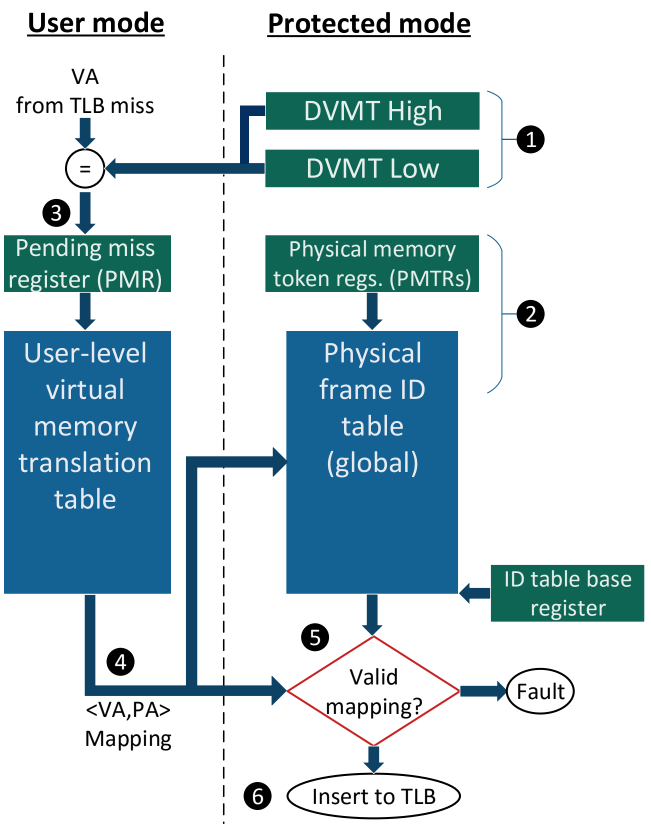

User mode
Protected mode
VA from TLB miss
DVMT High
DVMT Low
=
1
3
Pending miss register (PMR)
Physical memory token regs. (PMTRs)
2
User-level virtual memory translation table
Physical frame ID table (global)
ID table base register
4
5
<VA,PA> Mapping
Valid mapping?
Fault
6
Insert to TLB

Figure 5: Overall (simplified) flow of DVMT, including its major hardware, OS, and application components. The figure depicts hardware registers in green and software tables in blue. The registers are part of the process context and are replaced on context switches.

4 THE DVMT ARCHITECTURE

We start with an overview of how DVMT operates, including the hardware, OS, and application components. We then discuss the different components in detail in the case of an application running directly on the OS before describing how DVMT interacts with virtual machine workloads.

4.1 DVMT overview

Figure 5 depicts a simplified view of the overall DVMT flow; we elaborate on how the different steps are actually accomplished later in the section. ❶ When the application is initialized, it requests to use DVMT for a contiguous range of virtual addresses by setting the `DVMT_low` and `DVMT_high` bounds. These bounds are maintained by the OS as part of the process context and are used by the hardware to determine which TLB misses should use DVMT and which the standard page walk. ❷ Also during initialization, the process requests physical frames from the OS to be allocated in accordance with the DVMT scheme it employs (e.g., contiguous regions); note that DVMT supports arbitrary translation and dynamic allocation as well. The OS also creates the necessary permission tokens, which allow the system to enforce access permissions to physical frames independently from the translation process, and stores these tokens in the running process *physical memory token registers* (PMTRs). ❸ When a TLB miss occurs within the DVMT range, hardware triggers the translation sequence by updating a *pending miss register* (PMR) and waking up the DVMT helper thread. ❹ The helper thread performs the translation using its own state and returns the physical address to the hardware TLB. ❺ Before committing any accesses based on the translation, hardware validates that the translation does not violate memory protection. This is done by reading the permission token ID associated with the physical frame from the system global *physical frame ID table* in memory (pointed to by the *ID table base register*) and comparing it with the IDs in the process token registers. ❻ The TLB entry is then committed with the permissions from the matched token (tokens are only set and allocated by the OS).

In its simplest form, the DVMT architecture requires a minimum of four 64-bit registers and a 32-bit permission token register for each thread context; additional permission tokens may be added if shared memory regions are required. Adding extra pairs of bound registers allows for more than one DVMT region per-applications.

4.2 DVMT operation in native mode

We begin our description of DVMT architecture by describing its operation in native mode (i.e., no virtualization). Since DVMT breaks the common unification of *memory translation* and *memory isolation*, we explain the two separately, starting with protection and isolation, continuing through the translation process, and concluding the subsection by discussing necessary OS support.

4.2.1 **Enforcing Memory Protection and Isolation.** The DVMT architecture relies on permission tokens to ensure that the overall protection and isolation guarantees are enforced alongside features of conventional page-based virtual memory. The OS allocates each process a set of permission tokens. Each token consists of a unique ID and access permissions (read/write/execute). For permission tokens to work, the OS associates a process token with each allocated physical frame. Each frame may have just a single token, but multiple frames may share the same token. Each token thus identifies a region in which all frames share the same protection characteristics. In order for a process to access memory, it must have a token associated with the physical frame being accessed and the token permissions must allow the requested access. Notably, this design provides a simple capability system [29].

Process tokens are part of the process context and are stored in the hardware PMTRs when a process thread is active. All physical frame tokens are maintained in a system global *physical frame ID table*, which is stored in a contiguous region of physical memory that is managed by the OS. The table is accessed by hardware, which uses the accessed frame number to index the table and retrieve the frame's ID to check permissions. Access permissions are checked when a new translation is inserted into the TLB and when physical memory is accessed directly by the DVMT helper thread during translation. The permissions check is implemented in hardware. The frame's ID is retrieved from the physical frame ID table and is validated against the running thread's tokens (ID and permission) in the PMTRs.

The right-hand side of Figure 5 illustrates the access validation invoked when the translation thread inserts a virtual-to-physical mapping into the TLB. If the validation is successful, the mapping is committed to the TLB and the memory access that missed the TLB is re-issued. If the validation process fails, a page fault exception is raised to the OS. Notably, since the access permission validation is decoupled from the memory translation itself, the latency of the protection validation can be hidden using speculative execution [8]. Once the address translation completes, the processor can continue to execute in speculative mode until the protection checks complete as well.

The use of permission tokens guarantees that processes cannot access physical frames that belong to other processes unless they are explicitly shared when multiple processes are granted tokens with the same ID. Furthermore, processes cannot access or forge the contents of their permission tokens. Finally, the simplicity of the proposed mechanism allows access validation to be performed in hardware at the cost of a single memory access (retrieving a frame's ID from the physical frame ID table). It also allows for efficient caching of the contiguous frame table entries.

4.2.2 **VA $\rightarrow$ PA translation.** DVMT offloads the memory translation to a helper thread, which is triggered upon a TLB miss within the DVMT virtual address range reflected in the `DVMT_low` and `DVMT_high` registers. To avoid costly software context switches, the translation helper thread is co-scheduled with the application's execution thread on a separate (and potentially very lightweight) hardware thread. Once scheduled, the helper thread blocks until a translation is needed and does not interfere with the main thread. When a TLB miss occurs within the DVMT range, the memory management unit (MMU) writes the offending VA to the PMR to resume the blocked translation thread (non-DVMT addresses are translated using the system default page tables). The PMR serves as a simplified form of an M-Structure [9] and provides very fast communication and notification between the main and helper threads (as opposed to a costly trap to a miss handler used by OSs on MIPS [31, 32] and variants of SPARC [39]).

The helper thread performs the translation and may issue its own memory instructions. However, the memory addresses must either be outside the DVMT range or refer directly to physical addresses using special memory operations. This ability does not impose security risks because the architecture enforces physical memory isolation with the permission tokens. In order to support flexible translation schemes and take advantage of existing hardware, DVMT also provides special instructions that enable the translation thread to insert partial mappings to the PWCs and to query them. Once the PA is computed, the helper thread inserts it into the TLB using another special instruction. This operation then wakes up the main thread and the offending memory operation can complete or be re-issued for completion. The helper thread then immediately blocks and does not consume resources until the next time the PMR is written by the MMU. The new DVMT special instructions are summarized in Table 1.

Instruction	Description
insertTLB(⟨VA,PA⟩, perm)	Validate a ⟨VA, PA⟩ mapping and commit it into the DTLB with permissions *perms* extracted from the permission token.
insertPWC(CID,⟨VA,PA⟩,perm)	Validate a partial virtual memory mapping and commit it into the partial translation cache specified by *CID* with permissions *perms*.
lookupPWC(VA)	Look up the partial translation of *VA* in the all PWCs. Return the translation and the *CID* of the deepest partial translation cache that was hit.
loadPA(PA)	load the value stored in physical address *PA* (after validating it is accessible by the calling process).

Table 1: Listing of DVMT instructions.

Listing 1: Pseudocode of a simplified translation handler using DVMT{*flat*} translation scheme.

```
void TLBmiss_handler(void) {
  while(true) {
    // read offending VA from the PMR. operation
    // blocks until an address is available (TLB miss).
    VA = readPMR();

    // compute the offset of the offending VA's
    // page in the indirection table.
    MTB = mapping table physical base address
    offset = (VA - DVMT_Low) >> page_size_bits;

    // read the target frame pointer from the
    // indirection table.
    PA = loadPA(MTB + offset);

    // get the page permissions from the PMTR.
    perm = readPMTRPerms();

    if (PA != nullptr) {
      // the mapping exists; insert into TLB. operation
      // will fault if perm is more permissive than
      // the physical frame ID table.
      insertTLB( { VA, PA }, perm);
    } else {
      // the VA has not yet been mapped to a physical
      // frame. allocate a new frame.
      allocateFrame(VA);
    }
  }
}
```

4.2.3 **Custom memory translation schemes.** DVMT allows applications to customize their memory translation. In this paper we explore the following custom translations: *DVMT{*flat*}*, *DVMT{*hash*}*, and *DVMT{*DS*}*.

The *DVMT{*flat*}* scheme uses a simple flat indirection table. An array (contiguous in physical memory) stores the physical frame numbers and is indexed by virtual page number: $\lfloor \frac{virtual\,address - DVMT\_Low}{page\,size} \rfloor$. This scheme obtains a translation with a single memory access. Because the translation requires a contiguous array, the scheme best suites workloads with moderate-size memory requirements (up to a few GBs) and have a densely populated virtual address space. Listing 1 shows pseudo code of a translation handler for the DVMT{*flat*} translation scheme.

The *DVMT{*hash*}* scheme uses a hash table that functions as a software TLB and is backed by a 4-level software-managed radix tree. The number of buckets in the hash is configurable, and each entry holds 4 cache-line-aligned ⟨VA, PA⟩ pairs (the hash table is updated using a pseudo-LRU replacement policy). The hash table lookup is accelerated using the SIMD unit (512-bit wide, like that of the Intel Skylake). An entire 64B bucket is loaded into a vector register and its 4 entries are searched concurrently. If the hash table misses, the handler performs a software page walk. The walk is assisted by the hardware PWCs (which are the updated by the recovered mapping) and is therefore more susceptible to PWC flushes that occur on every context switch. This scheme obtains a translation with a single vector register load. The bounded hash table makes this scheme attractive for workloads with a huge virtual address space or a sparsely populated one.

Finally, *DVMT{*DS*}* imitates the direct segment translation introduced by Basu *et al.* [10] and extended by Gandhi *et al.* [23]. The scheme manages the DVMT virtual address range as a single, contiguous memory segment that maps directly to a physical memory segment. This scheme can translate addresses with a simple addition, but requires the use of a contiguous physical memory segment whose size matches that of the DVMT virtual range. Still, DVMT physical frame IDs allow the memory pages to be swapped out by the OS, which can change the frame ID such that the permission check fails leading to an exception; this is impossible in the original direct segment designs [10, 23].

4.2.4 **OS support.** In this section we discuss the OS modifications imposed by DVMT. Primarily, the OS must manage permission tokens and the physical frame ID table, as well as allow applications to explicitly allocate physical frames. In addition, the OS should allow translation threads to invoke TLB shootdowns, support swapping of frames that are managed by DVMT application code, and schedule the helper threads to hardware contexts.

Managing tokens and physical frames. The OS creates permission tokens that carry unique IDs, assigns tokens to processes, and associates frames with token IDs. It also manages the physical frame ID table, which stores all frame IDs. The OS allocates the physical frame token table at boot time. Whenever it allocates a physical frame for a process, it updates the frame ID with the designated process's permission token. When memory is shared, the OS creates a new token, associates its ID with the shared frames, and assigns the token to all the processes sharing the memory. Notably, managing the tokens at the OS level hides them from user code and prevents them from being forged.

User-level TLB shootdowns. Any change to the virtual-to-physical memory mapping must be propagated to all the TLBs in the system using TLB shootdowns. TLB shootdowns are (costly) software coherence scheme across TLBs that exists in all modern OSs as a kernel-level operation. Moving virtual-to-physical mappings to application code requires that TLB shootdowns can be invoked by the application code. This requires that the OS expose its TLB shootdown mechanism to DVMT processes through a dedicated system call.

Swapping and memory overcommit. The DVMT design preserves the paging property of modern virtual memory systems, which enables the OS to swap out unused memory pages. Whenever the OS swaps a physical frame to a secondary storage, it must also

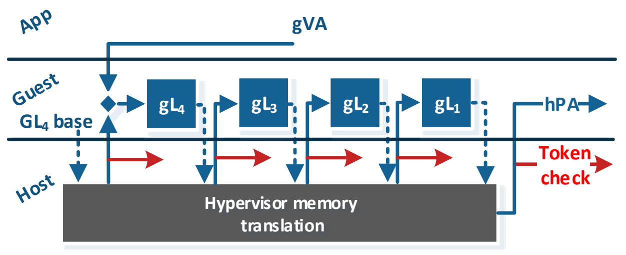


Figure 6: The translation flow when only the hypervisor uses the DVMT architecture.

signal the frame's owner process to update its user-level mapping tables. This is a key change that DVMT imposes on the OS swapping mechanism.

Importantly, security is not affected even when the application's memory manager ignores the signal. When swapping out a frame, the OS updates the frame's ID in the physical frame ID table and thereby prevents the process from using a stale mapping. Furthermore, before modifying the table, the OS must flush all mappings pertaining to the frame in all TLBs. This differs from existing TLB flush process in that entries are flushed based on the physical frame ID rather than page number, which requires some additional TLB logic.

Finally, managing memory translations at the application level makes it possible to optimize the selection of a swap victim. Instead of the OS generically determining which frames to swap out, it can allow the user process to judiciously select a victim frame (to avoid any security issues, an abort protocol [18] should be applied to overcome unresponsive applications).

Managing Accessed/Dirty bits. Legacy A/D bits are maintained in the physical frame ID table. Each entry stores a frame's permission token and its A/D bits, which are updated on TLB references.

4.3 Extending DVMT to virtualized workloads

Contemporary memory translation in virtual machines requires a 2D page walk to derive the hPA from a given gVA, and can benefit from customized translation. DVMT seeks to reduce the overhead in each dimension independently by enabling both host (hypervisor) and guest to manage their own memory translations. For brevity, we focus our discussion on two scenarios: (1) hypervisor-only DVMT (hVA-to-hPA); and (2) non-collaborative DVMT translation in both guest (gVA-to-gPA) and hypervisor (hVA-to-hPA).

Hypervisor-only DVMT. Hypervisor-only DVMT translation enables the hypervisor to manage the translation of the VM emulated physical memory region to host physical address (in the Linux KVM/QEMU model, this translation will be implemented by the QEMU VM-wrapper process). In this mode, translations inside the guest (gVA-to-gPA) are executed by the hardware page-walker and managed by the guest OS.

Figure 6 illustrates the hypervisor-only DVMT translation. For each level in the guest OS page tables, the hypervisor's custom translation thread executes a gPA-to-hPA translation. Naturally, in this mode only the hypervisor needs to be modified to incorporate DVMT, while guest OSs and applications remain unmodified.

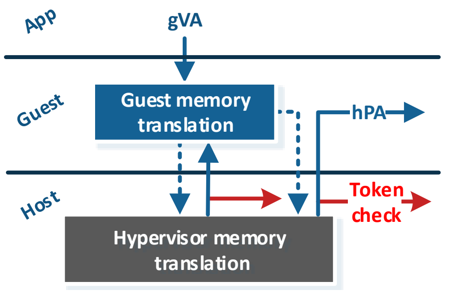


Figure 7: Virtual translation flow when both guest and hypervisor use DVMT architecture

DVMT in both guest and hypervisor. DVMT supports virtualized guests by replicating their hardware state (PMR, DVMT_High, DVMT_Low, PMTRs, and ID table base register) and providing guests VMs with a separate hardware context. This model is similar the support for nested pages tables in the x86-64 architecture [1]. The separate hardware context enables guest VMs to manage their own "physical" frame ID table (frames in guest physical address space) and to use DVMT hardware and let guest applications manage gVA-to-gPA memory translations. Notably, in guest mode, insertions to TLB and PWCs are validated both against the guest's PMTRs and the host's PMTRs.

Figure 7 illustrates the combined translation. The translation thread of the guest application translates its (guest) virtual addresses to (guest) physical address. The hypervisor then translates the guest physical address (host virtual) to a host physical address.

In summary, the DVMT architecture allows native and virtualized applications to customize the memory translation flow while imposing minimal hardware constraints. The following sections present our evaluation of the DVMT architecture.

5 METHODOLOGY

Running an instruction-level, full-system cycle-based simulator for virtualized workloads whose individual run time on a real system takes many minutes is not feasible. Instead, we model the performance impact of DVMT on memory translations by simulating the DVMT schemes described in Section 4.2.3 and comparing the translation overhead with detailed simulations of the MMU and the hardware page walker (including all TLBs and PWCs). Instead of comparing DVMT and Haswell MMU simulated times directly, we follow an approach that also takes into account an estimate of the translation overhead measured on an Intel Haswell system. We do this by estimating the ideal execution time (assuming perfect TLBs) by explicitly forcing hugepages and collecting performance counters data. We measure ideal time (T_{ideal}) and the baseline execution time with 4K pages ($T_{baseline}$). We also estimate the overhead of translation on the real system (OH_{HSW}). We then scale the overhead by the ratio of simulated DVMT and the simulated conventional MMU (the simulated overheads are Sim_{DVMT} and Sim_{HSW}, respectively) to estimate the measured translation overhead and compute speedup:

$$speedup_{DVMT} = \frac{T_{baseline}}{T_{ideal} + OH_{HSW} \times \frac{Sim_{DVMT}}{Sim_{HSW}}}$$

Benchmark	Description
Graph500	Generation, compression and BFS of large graphs
GraphLab, PowerGraph	Machine learning and data mining for graph analytics. Tunk Rank, on a 45GB data set of Twitter users
GUPS	Random memory accesses (HPC challenge)
memcached	In memory key-value cache, using 60GB scaled Twitter data set
mongoDB	Document database storing JSON-like documents with dynamic schemes (45GB scaled YCSB data set)
Redis	In-memory key-value DB (30GB YCSB dataset)
CG	Conjugate gradient benchmark from the NAS suite
canneal, dedup	PARSEC 3.0 compute benchmarks (native input set)
cactusADM, mcf	SPEC 2006 benchmarks (ref input)

Table 2: List of benchmarks

Processor	Dual socket Intel(R) Xeon(R) E5-2630v3 (HSW) @ 2.40GHz 8cores/socket, SMT
Memory	128GB DDR4 2133 MHz
Operating system	Ubuntu 14.04.3 LTS (kernel 3.12.13)
Hypervisor	2 cores QEMU version 2.5.5 with KVM
Guest OS	Ubuntu 14.04.3(kernel 3.19.0-25)
L1 DTLB	4-way, 64 entries, 1 cy. hit-to-data
L2 STLB	8-way, 1024 entries, 7 cy. hit-to-data
Partial Walk Caches	3-level partial walk caches, native and virtual (Intel parlance: PML4, PDP, PDE, ePML4, ePDP, ePDE)
L1 I/D Cache	32KB,8-way, 64B block, 4 cy. hit-to-data
L2 Cache	256KB, 8-way, 64B block, 12 cy. hit-to-data
L3 Cache	20MB, 8-way, 64B block, 25 cy. hit-to-data

Table 3: Experimental configuration

We argue that scaling the measured run times by simulating both DVMT and the Intel Haswell MMU is more representative of the potential speedup than directly approximating the run times using the simulated time (Sim_{mmu}) as $T_{ideal} + Sim_{mmu}$. The reason is that even if we perfectly simulate memory translations, looking only at translations cannot account for the subtle interactions between translation overheads, the executing cores (which may hide some of the translation latency with out-of-order execution), and the OS (that may invoke context switches and flush the MMU caches). Using the simulator results as a scaling factor thus puts both simulated models on equal footing and accounts for effects that are only observed on a real system. Importantly, the scaling methodology is a pessimistic approximation of DVMT speedup. For all benchmarks, scaling provided lower performance gains than those obtained with direct comparison of the simulated overheads alone.

The remainder of this section describes how we quantify the translation overhead on a real system and how we approximate the performance impact of DVMT. The benchmarks we used in this study are listed in Table 2. Notably, GUPS serves as a stress test due to its poor locality, which places near-maximal pressure on the memory architecture, including translation; GUPS accentuates translation overheads and allows for a way to consistently compare schemes.

Quantifying translation overheads on a real system.

We quantify the memory translation overheads by measuring applications run times on a real system (Table 3) and subtracting the approximated run time of an ideal system with perfect TLBs.

We approximate the run time on an ideal system by configuring the experimental platform to use huge 2MB pages with hugetlbfs, which increases the reach of the TLB to minimize the number of TLB misses without adding any of the management overheads associated with transparent huge pages. Note that Haswell also supports 1GB pages but only includes 4 TLB entries for them, which induces an excessive number of TLB misses. Because a few TLB misses still occur, we estimate the 2MB translation overhead using performance counters that measure the number of cycles in which the page walker (*page miss handler* in Intel parlance) was active. We deduct the 2MB translation overhead from the measured run time to estimate the ideal run time. Interestingly, even with 2MB pages the number of TLB misses was negligible, and the page walker was only active 1% of cycles on average. We note that this methodology is inspired by that used by Basu *et al.* [10] and by Gandhi *et al.* [23].

Simulation platform.

Our memory translation simulator replays translation traces (TLB misses) and models both DVMT and Haswell microarchitectures at the cycle level (including all TLBs and PWCs). We approximate the latency of Haswell's page table walker, TLB, STLB, PWCs, and data cache hierarchy by running a set of finely-tuned microbenchmarks on our experimental platform. We simulate the DVMT helper threads as running (pessimistically) in-order, and assume they can begin fetching instructions in the cycle following the write to the PMR. To model the memory access latencies of the page walker and DVMT, we collect page walk accesses hit rates in different levels of the cache hierarchy. We couple this information with a model for the latency of each level of the hierarchy.

The simulator is driven by TLB miss traces obtained using a modified version of BadgerTrap [22]. The traces capture virtual addresses that missed in all TLBs as well as the data cache hit rates tracked by the processor's performance counters [2]. The simulator determines which memory locations are accessed during translation for each TLB miss (for each mechanism, including any translation-specific caches). For each memory access required for translation, the simulator computes an access time. This is done by generating the cache level that the access will hit in (based on the traced performance counters statistics) and using the latency for that level (Table 3). For DRAM, we measured average memory access latency on our system and used a simple model for the hardware page walk and DVMT DRAM access: 200 cycles with a random fuzz factor of ± 20 cycles; we also experimented with other latencies (100 and 300 cycles) and did not observe substantial differences in the final speedup numbers. Finally, the simulator mimics the physical frame allocation policy used in the Linux kernel to map virtual pages to physical frames.

The simulated traces consist of a full run of the benchmark when possible. For extremely long runs or, alternatively, client-server benchmarks, we used traces consisting of at least 10^{10} instructions, which represents the benchmark's steady state running phase, and normalized the various configurations to the same amount of work. We use the performance counters to also estimate the overhead associated with translation-related cache misses (averaged over short time windows) and apply the same expected access times to the DVMT schemes.

6 EVALUATION

In this section we present the evaluation of the DVMT architecture using the translation schemes presented in Section 4.2.3. For DVMT{*hash*} we use a hashtable with 512K entries, or 8MB. We also compare with the Haswell translation scheme, which we denote

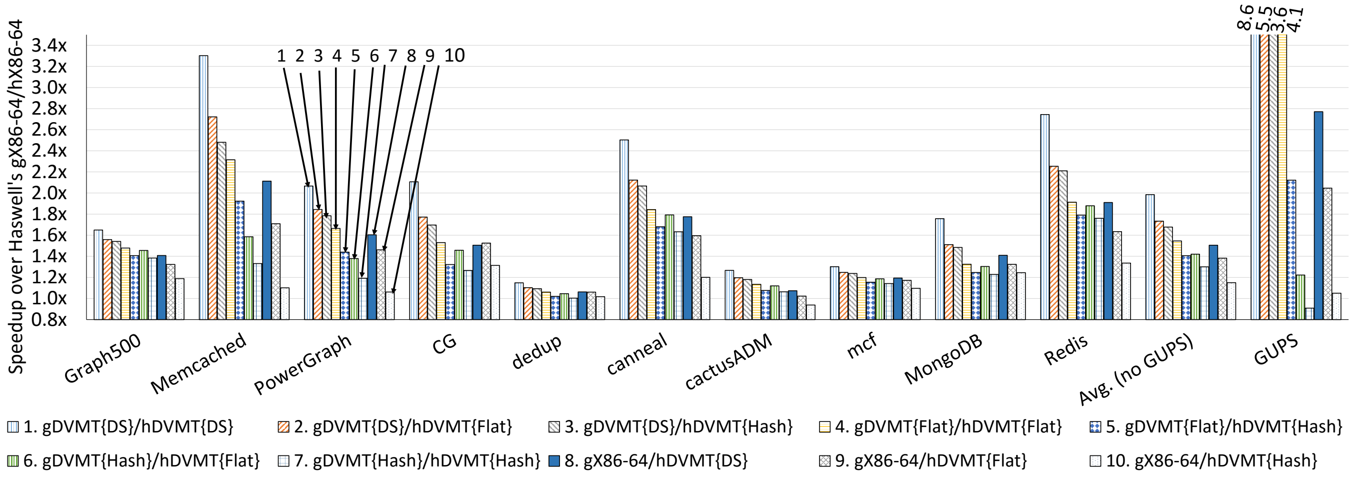


Figure 8: Speedups obtained with DVMT in virtualized environments over x86-64 virtualization (4K pages). The names of the translation schemes comprise both guest and host schemes as *guest/host*.

X86-64. We focus our discussion on the performance benefits of DVMT in virtualized environments, and then briefly discuss native environments. We conclude the evaluation with a short study of the hashtable miss rates.

6.1 Virtualized environments

We evaluate the performance of DVMT using different combinations of the possible translation schemes; we do not exhaustively investigate all possible combinations and focus on sensible ones. All methods use 4KB pages. A configuration that uses scheme A in the guest and B in the host is denoted as gA/hB (e.g., gX86-64/hDVMT{*hash*} indicates X86-64 translation in the guest and DVMT{*hash*} in the hypervisor).

Figure 8 depicts the speedups obtained by different guest/hypervisor configurations over the conventional gX86-64/hX86-64 translation scheme.

Running unmodified guest VMs. We begin by examining unmodified guest VMs with customized memory translation only in the hypervisor (configurations 8–10 in the figure). Figure 8 shows that applying DVMT only at the hypervisor level does not help applications whose virtual memory translations fit well in Haswell's large TLB and PWCs translation caches (e.g., dedup, mcf, cactusADM).

However, big-memory applications (e.g., PowerGraph, Memcached, Redis) that overflow the translation caches do benefit from DVMT's custom translation schemes. For example, both Memcached and Redis exhibit a speedup of $\sim1.6\times$ with gX86-64/hDVMT{*flat*} and $\sim2\times$ with gX86-64/hDVMT{*DS*}. PowerGraph lags slightly behind with a speedup of $\sim1.5\times$ with gX86-64/hDVMT{*flat*} and $\sim1.6\times$ with gX86-64/hDVMT{*DS*}. The direct-segment configuration, however, requires that the VM emulated memory reside in contiguous physical frames, which is not always possible.

On average, we see that for unmodified guests using DVMT{*flat*} at the hypervisor level outperforms the DVMT{*hash*} (average speedups of $1.4\times$ vs. $1.2\times$). This is expected since the indirection array in DVMT{*flat*} acts as a perfect hashtable. However, the flat table requires significantly more memory capacity as it scales linearly with application allocation compared to the fixed-size hashtable. Some applications (e.g., MongoDB) do not benefit significantly from "upgrading" from a hash to a flat table in the host and can save memory capacity and perhaps increase cache hit rates for the hashtable accesses. This demonstrates the benefits of the flexible approaches made possible by software "doing it itself". We discuss this more in Section 6.3.

Custom translation in both guest and host. We now turn to examine the benefits of DVMT for custom translation in both the host and the guest. As discussed in Section 4.2.3, the fastest yet most restrictive translation scheme is DVMT{*DS*}. Indeed, applying the scheme to both guest and host in gDVMT{*DS*}/hDVMT{*DS*} delivers the best speedups — $3.3\times$ for memcached, $2.5\times$ for canneal, $1.8\times$ for MongoDB, $2.8\times$ for Redis, and a whopping $8.6\times$ for GUPS. On average, this configuration delivers $2\times$ speedup.

Memory management flexibility is crucial for hypervisors and DVMT{*DS*} is not a realistic choice as it requires contiguous physical memory segment which interferes with memory utilization techniques such as memory overcommit and deduplication. At the same time, these hypervisor-level utilization techniques do make DVMT{*DS*} a reasonable choice for guest VMs. Guests may control their own physical memory in a simple way and rely on the host to optimize physical memory usage.

The gDVMT{*DS*}/hDVMT{*flat*} and gDVMT{*DS*}/hDVMT{*hash*} DVMT configurations (configurations 2 and 3 in Figure 8) provide the hypervisor with much greater flexibility to manage system memory. The big-memory applications such as *Memcached*, *MongoDB*, and *Redis* still exhibit substantial speedups when using flexible schemes at the hypervisor level. gDVMT{*DS*}/hDVMT{*flat*} delivers speedups of $2.7\times$, $1.5\times$, and $2.3\times$ for these applications, respectively. gDVMT{*DS*}/hDVMT{*hash*} delivers almost similar performance — $2.5\times$, $1.5\times$, and $2.2\times$, respectively. Even PWC-friendly workloads such as *Dedup*, *cactusADM*, and *mcf* benefit from those configurations with speedups $1.1\times$, $1.2\times$, and $1.3\times$, respectively, or higher.

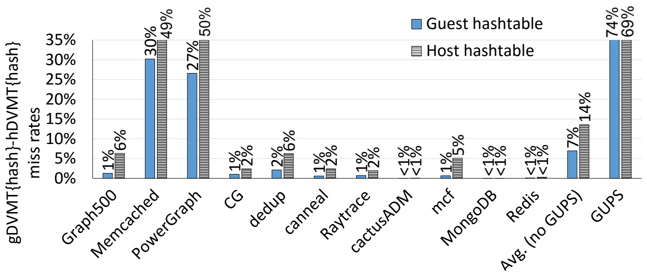


Figure 9: Miss rates in gDVMT{*hash*}/hDVMT{*hash*} with 512K entries each.

On average, we see that gDVMT{*DS*}/hDVMT{*flat*} and gD-VMT{*DS*}/hDVMT{*hash*} gain ∼1.7× speedup. We also see that the other configurations (indexed 4–7 in the figure) deliver average speedups of 1.3–1.6×.

In summary, our evaluation demonstrates the performance benefits of custom address translation in both guest VM and host. As expected the best performing translation scheme is the rigid DV-MT{*DS*}, but we also see that the alternative and more feasible DVMT{*flat*} and DVMT{*hash*} provide good alternatives. As with the X86-64 guest, we see that some applications are very sensitive to the choice between a more expensive flat table and a cheaper hashtable, while others are not.

6.2 Native environments

Using DVMT for native translation (i.e., without virtualization) provides little benefit. As Figure 3 shows, the TLB and PWC of Haswell greatly reduce the overhead of native virtual memory translation and leave little room for improvement. Our experiments show that D-VMT{*flat*} and DVMT{*hash*} outperform x86-64 by up to 4% on average, out of the total 12% translation overheads that x86-64 suffers from. DVMT{*DS*} eliminates the translation overheads altogether but imposes major restrictions on memory management. Importantly, no DVMT scheme underperforms x86-64.

6.3 Hashtable missrates

Figure 9 shows the hashtable miss rates at both the guest and host levels when using gDVMT{*hash*}/hDVMT{*hash*}, each with 512K entries. Most applications exhibit negligible miss rates even with a hashtable that consumes just 8MB of memory. A few of the large irregular applications, however, may benefit from larger hashtables and exhibit high missrates. This explains the greater sensitivity these applications show to the DVMT scheme because all hashtable misses proceed as x86-64 TLB misses. It is also interesting to note that the miss rate at the host is higher than that of the guest (with the exception of GUPS). This is because each miss in the guest generates two host accesses — one for translating the hashtable miss and the other for translating the hashtable access itself; thus placing greater pressure on the host hashtable. GUPS has completely random memory access patterns and the miss rates are so high that they do not follow the expected trend; the relative difference between them is small.

7 CONCLUSIONS

Do-It-Yourself Virtual Memory Translation (DVMT) is an architecture for applications to customize their virtual-to-physical address translation. DVMT facilitates translation customization by decoupling memory protection validation from virtual-to-physical mapping. Memory protection is implemented with a lightweight capability system that is managed by the OS and invoked by hardware. This decoupling allows delegating the virtual-to-physical translation to an application helper thread that is co-scheduled with its main thread. The helper thread wakes upon TLB misses using a lightweight hardware synchronization mechanism.

We show that DVMT is particularly effective in virtualized environments, in which translation overhead is higher. DVMT supports memory optimization methods commonly used in hypervisors, including paging, memory overcommit, and deduplication. We show that various custom translation schemes speed up a number of big-memory virtualized workloads by 1.2–2.0×.

We note that the DVMT translation thread effectively embodies a programmable page table walker. Since we opted to focus this paper on motivating custom virtual memory translation and showing its performance benefits, DVMT uses a common multithreaded processor design in which the translation thread runs on a separate hardware context. A more sophisticated implementation can run the translation thread on a small, programmable core that will replace prevalent fixed hardware page table walkers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We also thank Lluìs Vilanova, Muli Ben-Yehuda, and Joseph Nuzman for their insightful comments and feedback on the paper. This research was funded by the Israel Science Foundation (ISF grant 769/12) and the Israel Ministry of Science, Technology, and Space.

REFERENCES

[1] 2016. *Intel© 64 and IA-32 Architectures Software Developer's Manual*.
[2] 2016. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. (2016).
[3] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*. https://doi.org/10.1145/1168857.1168860
[4] Advanced Micro Devices 2015. *AMD64 Architecture Programmer's Manual (Volume 2)*. Advanced Micro Devices.
[5] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting hardware-assisted Page Walks for virtualized systems. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/2366231.2337214
[6] ARM 2016. *ARMv8 Architecture Reference Manual*. ARM.
[7] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: skip, don't walk (the page table). In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/1815961.1815970
[8] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: a mechanism for speculative address translation. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/2000064.2000101
[9] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. 1991. M-Structures: extending a parallel, non-strict, functional language with state. In *ACM Conf. on Functional Programming Languages and Computer Architecture*. https://doi.org/10.1007/3540543961_26
[10] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient virtual memory for big memory servers. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/2485922.2485943
[11] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*. https://doi.org/10.1145/1346281.1346286

[12] Abhishek Bhattacharjee. 2013. Large-reach memory management unit caches. In *Intl. Symp. on Microarchitecture (MICRO)*. https://doi.org/10.1145/2540708.2540741

[13] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *Symp. on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2011.5749717

[14] Abhishek Bhattacharjee and Margaret Martonosi. 2009. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Intl. Conf. on Parallel Arch. and Compilation Techniques (PACT)*. https://doi.org/10.1109/PACT.2009.26

[15] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and RH Taheri. 2013. Methodology for performance analysis of VMware vSphere under Tier-1 applications. *VMware Technical Journal* 2, 1 (2013).

[16] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. 2013. Improving virtualization in the presence of software managed translation lookaside buffers. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/2485922.2485933

[17] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. 1999. Simultaneous subordinate microthreading (SSMT). In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.1999.765950

[18] Dawson Engler, Frans Kaashoek, and James O'Toole, Jr. 1995. Exokernel: an operating system architecture for application-level resource management. In *ACM Symp. on Operating Systems Principles (SOSP)*. https://doi.org/10.1145/224056.224076

[19] Dawson R. Engler, Sandeep K. Gupta, and Frans M. Kaashoek. 1995. AVM: Application-level virtual memory. In *Hot Topics in Operating Systems (HotOS)*. https://doi.org/10.1109/HOTOS.1995.513458

[20] Zhen Fang, Lixin Zhang, John B Carter, Wilson C Hsieh, and Sally A McKee. 2001. Reevaluating online superpage promotion with hardware support. In *Symp. on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2001.903252

[21] Narayanan Ganapathy and Curt Schimmel. 1998. General purpose operating system support for multiple page sizes. In *USENIX Ann. Tech. Symp. (ATC)*. http://dl.acm.org/citation.cfm?id=1268256.1268264

[22] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *Computer Architecture News* 42, 2 (Sept. 2014), 20–23. https://doi.org/10.1145/2669594.2669599

[23] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient memory virtualization: reducing dimensionality of nested page walks. In *Intl. Symp. on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2014.37

[24] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile paging: exceeding the best of nested and shadow paging. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2016.67

[25] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively breaking large pages to improve memory overcommitment performance in VMware ESXi. In *Intl. Conf. on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/2731186.2731187

[26] Bruce Jacob and Trevor Mudge. 1997. Software-managed address translation. In *Symp. on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.1997.569652

[27] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the distance for TLB prefetching: an application-driven study. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2002.1003578

[28] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2016. Energy-efficient address translation. In *Symp. on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2016.7446100

[29] Henry M Levy. 1984. *Capability-based computer systems*. Digital Press.

[30] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs. *ACM Trans. on Arch. & Code Optim.* 10, 1 (2013). https://doi.org/10.1145/2445572.2445574

[31] MIPS Technologies 2011. *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture*. MIPS Technologies. Revision 3.02.

[32] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. 1993. Design tradeoffs for software-managed TLBs. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/165123.165127

[33] Binh Pham, Arup Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *Symp. on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2014.6835964

[34] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large pages and lightweight memory management in virtualized environments: can you have it both ways?. In *Intl. Symp. on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830773

[35] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: A case for software address translation on GPUs. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2016.58

[36] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB performance of superpages with less operating system support. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems (ASPLOS)*. https://doi.org/10.1145/195473.195531

[37] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in supporting two page sizes. In *Intl. Symp. on Computer Architecture (ISCA)*. https://doi.org/10.1145/139669.140406

[38] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. 2011. Selective hardware/software memory virtualization. In *Intl. Conf. on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/1952682.1952710

[39] David L. Weaver and Tom Germond (Eds.). 1994. *The SPARC Architecture Manual (Version 9)*. Prentice Hall. SPARC International, Inc.

[40] Timothy Wood, Gabriel Tarasuk-levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. 2009. Memory Buddies: exploiting page sharing for smart colocation. In *Intl. Conf. on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/1508293.1508299

[41] Idan Yaniv and Dan Tsafrir. 2016. Hash, don't cache (the page table). In *Intl. Conf. on Measurement & Modeling of Computer Systems (SIGMETRICS)*. https://doi.org/10.1145/2896377.2901456

[42] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. 2010. Enigma: architectural and operating system support for reducing the impact of address translation. In *ACM Intl. Conf. on Supercomputing*. https://doi.org/10.1145/1810085.1810109