Hardware Translation Coherence for Virtualized Systems

Zi Yan Ján Veselý Guilherme Cox Abhishek Bhattacharjee
Department of Computer Science, Rutgers University
{zi.yan,jan.vesely,guilherme.cox,abhib}@cs.rutgers.edu

ABSTRACT

To improve system performance, operating systems (OSes) often undertake activities that require modification of virtual-to-physical address translations. For example, the OS may migrate data between physical pages to manage heterogeneous memory devices. We refer to such activities as page remappings. Unfortunately, page remappings are expensive. We show that a big part of this cost arises from address translation coherence, particularly on systems employing virtualization. In response, we propose **hardware translation invalidation and coherence** or **HATRIC**, a readily implementable hardware mechanism to piggyback translation coherence atop existing cache coherence protocols. We perform detailed studies using KVM-based virtualization, showing that HATRIC achieves up to 30% performance and 10% energy benefits, for per-CPU area overheads of 0.2%. We also quantify HATRIC's benefits on systems running Xen and find up to 33% performance improvements.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Virtual machines**; **Operating systems**; **Virtual memory**;

KEYWORDS

Virtualization, translation coherence, heterogeneous memory.

ACM Reference format:
Zi Yan Ján Veselý Guilherme Cox Abhishek Bhattacharjee Department of Computer Science, Rutgers University . 2017. Hardware Translation Coherence for Virtualized Systems. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080211

1 INTRODUCTION

As the computing industry designs systems for big-memory workloads, system architects have begun embracing heterogeneous memory architectures. For example, Intel is integrating high-bandwidth on-package memory in its Knight's Landing chip and 3D Xpoint memory in several products [29]. AMD and Hynix are releasing High-Bandwidth Memory or HBM [14, 35]. Similarly, Micron's Hybrid Memory Cube [48, 59] and byte-addressable persistent memories [18, 54, 67, 68] are quickly gaining traction. Vendors are combining these high-performance memories with traditional high capacity and low cost DRAM, prompting research on heterogeneous memory architectures [2, 5, 35, 43, 46, 49, 54, 64].

Fundamentally, heterogeneous memory management requires that OSes remap pages between memory devices with different latency / bandwidth / energy characteristics for desirable overall operation. Page remapping is not a new concept. OSes have long used it to migrate physical pages to defragment memory and create superpages [4, 36, 45, 63], to migrate pages among NUMA sockets [25, 37], and to deduplicate memory by enabling copy-on-write optimizations [52, 53, 58]. However, while page remappings are used sparingly in these scenarios, they become frequent when using heterogeneous memories. This is because page remapping is necessary for applications to utilize a memory device's technology characteristics by moving data to these memory devices. Consequently, IBM and Redhat are already deploying Linux patchsets to enable page remapping amongst coherent heterogeneous memory devices [16, 26, 34].

These efforts face an obstacle: page remappings suffer performance and energy penalties. There are two components to these penalties. The first is the overhead of copying data. The second is the cost of translation coherence. It is this second cost that this paper focuses on. When privileged software remaps a physical page, it has to update the corresponding virtual-to-physical page translation in the page table. Translation coherence is the means by which caches dedicated to translations (e.g., TLBs [12, 40, 50, 51], MMU caches [10], etc.) are kept up-to-date with page table mappings.

Past work has shown that translation coherence overheads can consume 10-30% of system runtime [46, 57, 65]. These overheads are even worse on virtualized systems. We show that as much as 40% of application runtime on virtualized systems can be wasted on translation coherence overhead. This is because modern virtualization support requires the use of two page tables. Systems with hardware assists for virtualization like Intel VT-x and AMD-V use a guest page table to map guest virtual pages to guest physical pages and a nested page table to map guest physical pages to system physical pages [9]. Changes to either page table require translation coherence.

The problem of coherence is not restricted to translation mappings. In fact, the systems community has studied problems posed by cache coherence for several decades [61] and has developed efficient hardware cache coherence protocols [42]. What makes translation coherence challenging today is that unlike cache coherence, it relies on cumbersome software support. While this may have sufficed in the past when page remappings were used relatively infrequently, it is problematic today as heterogeneous memories require more frequent page remapping. Consequently, we believe that there is a need to architect better support for translation coherence. In order to understand what this support should constitute, we list three attributes desirable for translation coherence.

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/10.1145/3079856.3080211

① **Precise invalidation:** Processors use several hardware translation structures – TLBs, MMU caches [7, 10], and nested TLBs (nTLBs) [9] – to cache portions of the page table(s). Ideally, translation coherence should invalidate the translation structure entries corresponding to remapped pages, rather than flushing all the contents of these structures.

② **Precise target identification:** The CPU running privileged code that remaps a page is known as the *initiator*. An ideal translation coherence protocol would allow the initiator to identify and alert only CPUs whose TLBs, MMU caches, and nTLBs cache the remapped page's translation. By restricting coherence messages to only these *targets*, other CPUs remain unperturbed by coherence activity.

③ **Lightweight target-side handling:** Target CPUs should invalidate their translation structures and relay acknowledgment responses to the initiator quickly, without excessively interfering with workloads executing on the target CPUs.

Over time, vendors have addressed some of these requirements. For example, x86-64 and ARM architectures support instructions that invalidate specific TLB entries, obviating the need to flush the entire TLB in some cases. OSes like Linux can track coherence targets (though not with complete precision so some spurious coherence activity remains) [65]. Crucially however, all this support is restricted to native execution. Translation coherence *for virtualized systems* meets none of these goals today.

In particular, virtualized translation coherence becomes especially problematic when there are changes to the nested page table. Consider ① – when hypervisors change a nested page table entry, they track guest physical and system physical page numbers, but not the guest virtual page. Unfortunately, x86-64 and ARM only allow precise TLB invalidation for entries whose guest virtual page is known. Consequently, hypervisors are forced to conservatively flush all translation structures, even if only a single page is remapped. This degrades performance since virtualized systems need expensive two-dimensional page table walks to re-populate the flushed structures [3, 9, 11, 15, 17, 24, 50, 51, 53].

Virtualized translation coherence protocols also fail to achieve ②. Hypervisors track the subset of CPUs that a guest VM runs on but cannot (easily) identify the CPUs used by a process within the VM. Therefore, when the hypervisor remaps a page, it conservatively initiates coherence activities on all CPUs that may potentially have executed *any* process in the guest VM. While this does spare CPUs that never execute the VM, it needlessly flushes translation structures on CPUs that execute the VM but not the process.

Finally, ③ is also not implemented. Initiators currently use expensive inter-processor interrupts (on x86) or tlbi instructions (on ARM, Power) to prompt VM exits on all target CPUs. Translation structures are flushed on a VM re-entry. VM exits are particularly detrimental to performance, interrupting the execution of target-side applications [1, 9].

We believe that it is time to implement translation coherence in hardware to solve these issues. This view is inspired by influential prior work on UNITD [57], which showcased the potential of hardware translation coherence. We propose **hardware translation invalidation and coherence** or **HATRIC**, a hardware mechanism that goes beyond UNITD and other recent work on TLB coherence for native systems [46, 65], and tackles ①-③. HATRIC extends translation structure entries with coherence tags (or co-tags) storing the system physical address where the translation entry resides (not to be confused with the physical address stored in the page table). This solves ①, since translation structures can now be identified by the hypervisor *without* knowledge of the guest virtual address. HATRIC exposes co-tags to the underlying cache coherence protocol, achieving ② and ③.

We evaluate HATRIC under a forward-looking virtualized system with a high-bandwidth die-stacked memory and a slower off-chip memory. HATRIC improves performance by up to 33% and saves up to 10% of energy, but requires only 0.2% additional CPU area. Overall, our contributions are:

- We quantify the overheads of translation coherence on hypervisor-managed die-stacked memory. We focus on KVM but also study Xen. All prior work on translation coherence [46, 57, 65] overlooks the problems posed by changes to nested page tables. We show that such changes cause slowdown, but that better translation coherence can potentially improve performance by as much as 35%.
- We design HATRIC to subsume translation coherence in hardware by piggybacking on existing cache coherence protocols. Our initial goal was to use UNITD, with the simple extensions recommended in the original paper [57] for virtualization. However, we found UNITD to be inadequate for virtualization in three important ways. First, UNITD (and indeed all prior work on translation coherence [46, 65]) ignores MMU caches and nested TLBs, which we find accounts for 8-15% of system runtime. Second, UNITD requires large energy-hungry CAMs. Third, the original UNITD work presents a blueprint, but not concrete details, on how to fold translation coherence atop directory-based coherence protocols. HATRIC addresses all three shortcomings to provide a complete end-to-end solution for virtualized translation coherence.
- We perform several studies that illustrate the benefits of HATRIC's design decisions. Further, we discuss HATRIC's advantages over purely software approaches to mitigate translation coherence issues.

While we focus mostly on the particularly arduous challenges of translation coherence due to nested page table changes, HATRIC is also applicable to shadow paging [3, 24] and native execution.

2 BACKGROUND

We begin by presenting an overview of the key hardware and software structures involved in page remapping. Our discussion focuses on x86-64 systems. Other architectures are broadly similar but differ in some low-level details.

2.1 HW and SW Support for Virtualization

Virtualized systems accomplish virtual-to-physical address translation in one of two ways. Traditionally, hypervisors used shadow page tables to map guest virtual pages (GVPs) to system physical pages (SPPs), keeping them synchronized with guest OS page tables [3].

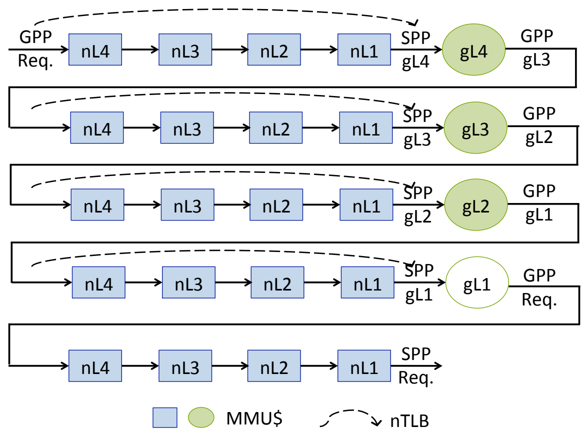

Figure 1: Two-dimensional page table walks for virtualized systems. Nested page tables are represented by boxes and guest page tables are represented by circles. Each page table's levels from 4 to 1 are shown. We show items cached by MMU caches and nTLBs. TLBs (not shown) cache translations from the requested guest virtual page (GVP) to the requested system physical page (SPP).

However, the overheads of page table synchronization can be high [24]. Consequently, most systems now use two-dimensional page tables instead. Figure 1 illustrates two-dimensional page table walks (see past work for more details [3, 7–9, 23, 53]). Guest page tables map GVPs to guest physical pages (GPPs). Nested page tables map GPPs to SPPs. Further, x86-64 systems use 4-level forward mapped radix trees for both page tables [8, 9, 23, 53]. We refer to these as levels 4 (the root level) to 1 (the leaf level) similar to recent work [7, 9, 10]. When a process running in a guest VM makes a memory reference, its GVP is translated to an SPP. The guest CR3 register is combined with the requested GVP (not shown in the picture) to deduce the GPP of level 4 of the guest page table (shown as GPP Req.). However, to look up the guest page table (gL4-gL1), the GPP must be converted into the SPP where the page table resides. Therefore, we first use the GPP to look up the nested page tables (nL4-nL1), to find SPP gL4. Looking up gL4 then yields the GPP of the next guest page table level (gL3). The rest of the page table walk proceeds similarly, requiring 24 sequential, and hence expensive, memory references in total. CPUs use three types of translation structures to accelerate this walk:

① **Private per-CPU TLBs** cache the requested GVP to SPP mappings, short-circuiting the entire walk. TLB misses trigger hardware page table walkers to look up the page table.

② **Private per-CPU MMU caches** store intermediate page table information to accelerate parts of the page table walk [7, 9, 10]. There are two flavors of MMU cache. The first is a *page walk cache* and is implemented in AMD chips [9, 10]. Figure 1 shows the information cached in page walk caches. Page walk caches are looked up with GPPs and provide SPPs where page tables are stored. The second is called a *paging structure cache* and is implemented by Intel [7, 10]. Paging structure caches are looked up with GVPs and provide the SPPs of page table locations. Paging structure caches generally perform better, so we focus on them [7, 10].

③ **Private per-CPU nTLBs** short-circuit nested page table lookups by caching GPP to SPP translations [9]. Figure 1 shows the information cached by nTLBs.

Apart from caching translations in these dedicated structures, CPUs also cache page tables in private L1 (L2, etc.) caches and the shared last-level cache (LLC). The presence of separate private translation caches poses coherence problems. While standard cache coherence protocols ensure that page table entries in private L1 (L2, etc.) caches are coherent, there are no such guarantees for TLBs, MMU caches, and nTLBs. Instead, privileged software keeps translation structures coherent with data caches and one another.

2.2 Page Remapping in Virtualized Systems

We now detail how translation coherence can be triggered on a virtualized system. All page remappings can be classified by the data they move, and the software agent initiating the move.

Remapped data: Systems may remap a page storing ⓐ the guest page table; ⓑ the nested page table; or ⓒ non-page table data. Most remappings are from ⓒ as they constitute most memory pages. We have found that less than 1% of page remappings correspond to ⓐ-ⓑ. We therefore highlight HATRIC's operation using ⓒ although HATRIC also implicitly supports the first two cases.

Remapping initiator: Pages can be remapped by a guest OS or the hypervisor. When a guest OS remaps a page, the guest page table changes. Past work achieves low-overhead guest page table coherence with simple and effective software approaches [47]. Unfortunately, there are no such workarounds to mitigate the translation coherence overheads of hypervisor-initiated nested page table remappings. For these reasons, cross-VM memory deduplication [27, 53] and page migration between NUMA memories on multi-socket systems [6, 55, 56] are known to be expensive. In the past, such overheads may have been mitigated by using these optimizations sparingly. However, with heterogeneous memories such as die-stacked memory, we may actually *desire* nested page table remappings to dynamically migrate data for good performance (see past work exploring paging policies for die-stacked memory [46]).

3 SOFTWARE TRANSLATION COHERENCE

Our goal is to ensure that translation coherence does not impede the adoption of heterogeneous memories. We study forward-looking die-stacked DRAM as an example of an important heterogeneous memory system. Die-stacked memory uses DRAM stacks that are tightly integrated with the processor die using high-bandwidth links like through-silicon vias or silicon interposers [32, 46]. Die-stacked memory is expected to be useful for multi-tenant and rack-scale computing where memory bandwidth is often a performance bottleneck and will require a combination of application, guest OS, and hypervisor management [20, 38, 46, 66].

3.1 Translation Coherence Overheads

We quantify translation coherence overheads on a die-stacked system that is virtualized with KVM. We modify KVM to page between the die-stacked and off-chip DRAM. Since ours is the first work to consider hypervisor management of die-stacked memory, we implement

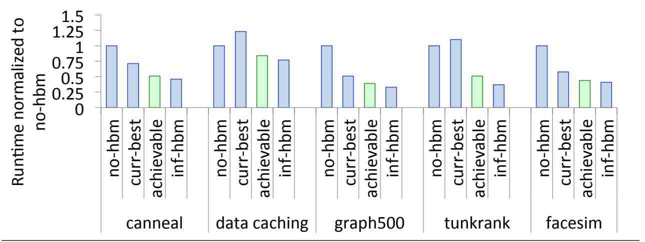

Figure 2: Performance of no-hbm **(no die-stacked DRAM),** inf-hbm **(data always in die-stacked DRAM),** curr-best **(best die-stacked DRAM paging policy with current software translation coherence overheads), and** achievable **(best paging policy, assuming no translation coherence overheads). Data is normalized to** no-hbm **runtime.**

a variety of paging policies. Rather than focusing on developing a single best policy, our objective is to show that current translation coherence overheads are so high that they generally curtail the effectiveness of any paging policy.

Our paging mechanisms extend prior work on software-guided die-stacked DRAM paging [46]. When off-chip DRAM data is accessed, there is a page fault. KVM then migrates the desired page into an available die-stacked DRAM physical page frame. The GVP and GPP remain unchanged but KVM changes the SPP and hence its nested page table entry. This triggers translation coherence.

We run our modified KVM on the detailed cycle-accurate simulator described in Section 5. Like prior work [46], we model a system with 2GB of die-stacked DRAM with 4× the memory bandwidth of a slower off-chip 8GB DRAM. This is a total of 10GB of addressable DRAM. We model 16 CPUs based on Intel's Haswell architecture.

Figure 2 quantifies the performance of hypervisor-managed die-stacked DRAM. We normalize all performance numbers to the runtime of a system with only off-chip DRAM and no high-bandwidth die-stacked DRAM (no-hbm). Further, we show an unachievable best-case scenario where all data fits in an infinite-sized die-stacked memory (inf-hbm). After profiling several paging strategies (evaluated in detail in Section 6), we plot the best-performing ones with the curr-best bars. These results assume traditional software translation coherence mechanisms. In contrast, the achievable bars represent the potential performance of the best paging policies with ideal zero-overhead translation coherence.

Figure 2 shows that (unachievable) infinite die-stacked DRAM can improve performance by 25-75% (inf-hbm versus no-hbm). Unfortunately, the current best paging policies (curr-best) fall far short of ideal inf-hbm. Translation coherence overheads are a big culprit – when these overheads are eliminated in achievable, system performance comes within 3-10% of the case with infinite die-stacked DRAM capacity (inf-hbm). In fact, Figure 2 shows that translation coherence overheads can be so high that they can prompt die-stacked DRAM to, counterintuitively, *degrade* performance. For example, data caching and tunkrank suffer 23% and 10% performance degradations in curr-best, respectively, despite using high-bandwidth die-stacked memory.

We also compare the costs of translation coherence to those of the actual data copy. We find that translation coherence can degrade performance almost as badly as data copying. For example, when running canneal with 16 CPUs, both translation coherence and data copying consume 30% of runtime. Like us, others have also noted that translation coherence can surprisingly exceed or match data copy costs [46]. Intuitively, this is because translation coherence scales poorly compared to data copying. While copying involves reading and writing a fixed-size page's data contents between memories regardless of core counts, translation coherence costs continue to increase with more cores. Virtualization exacerbates this problem by forcing VM exits on all these cores.

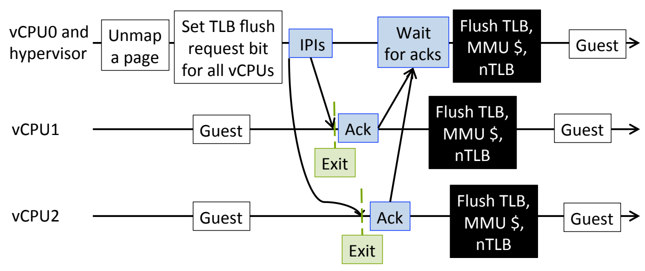

Figure 3: Sequence of operations associated with a page unmap. Initiator to target IPIs are shown in blue ①, VM exits are shown in green ②, and translation structure flushes are shown in black ③.

3.2 Page Remapping Anatomy

We now shed light on why translation coherence performs poorly. While we use page migration between off-chip and die-stacked DRAM as our driving example, the same mechanisms are used today to migrate pages between NUMA memories, or to defragment memory. When a VM is configured, KVM assigns it virtual CPU threads or vCPUs. Figure 3 assumes 3 vCPUs executing on physical CPUs. Suppose vCPU 0 frequently demands data in GVP 3, which maps to GPP 8 and SPP 5, and that SPP 5 resides in off-chip DRAM. The hypervisor may want to migrate SPP 5 to die-stacked memory (e.g. SPP 512) to improve performance. On a VM exit (assumed to have occurred prior in time to Figure 3), the hypervisor modifies the nested page table to update the SPP, triggering translation coherence. There are three problems with this:

All vCPUs are identified as targets: Figure 3 shows that the hypervisor initiates translation coherence by setting the TLB flush request bit in every vCPU's kvm_vcpu structure. The kvm_vcpu structure stores vCPU state. When a vCPU is scheduled on a physical CPU, kvm_vcpu is used to provide register content, instruction pointers, etc. By setting bits in kvm_vcpu, the hypervisor signals that TLB, MMU cache, and nTLB entries need to be flushed.

Ideally, we would like the hypervisor to identify only the CPUs that cache the stale translation as targets. The hypervisor does skip physical CPUs that never executed the VM. However, it flushes all physical CPUs that ran any of the vCPUs of the VM, regardless of whether they cache the modified page table entries.

All vCPUs suffer VM exits: Next, the hypervisor launches inter-processor interrupts (IPIs) to all the vCPUs. IPIs are deployed by the processor's advanced programmable interrupt controllers (APICs). APIC implementations vary and depending on the APIC technology, KVM converts broadcast IPIs into a loop of individual IPIs or a loop across processor clusters. We have profiled IPI overheads using microbenchmarks on Haswell systems and like past work [46, 65], we find that they consume thousands of clock cycles. If the receiving CPUs are running vCPUs, they suffer VM exits, compromising goal ③ from Section 1. IPI targets then acknowledge the initiator, which is paused waiting for all vCPUs to respond.

All translation structures are flushed: The next step is to invalidate stale mappings in translation structure entries. Current architectures provide ISA and microarchitectural support for this via, for example, invlpg instructions in x86. There are two caveats however. First, these instructions need the GVP of the modified nested page table mapping to identify the TLB entries that need to be invalidated. This is primarily because modern TLBs maintain GVP bits in the tag. While this is a good design choice for non-virtualized systems, it is problematic for virtualized systems because hypervisors do not have easy access to GVPs. Instead, they have GPPs and SPPs. Consequently, KVM and Xen flush all TLB contents when they modify a nested page table entry, rather than selectively invalidating TLB entries. Second, there are currently no instructions to selectively invalidate MMU caches or nTLBs, even though they are tagged with GPPs and SPPs. This is because the marginal benefits of adding ISA support for selective MMU cache and nTLB invalidation are limited when the more performance-critical TLBs are flushed.

3.3 Hardware Versus Software Solutions

It is natural to ask whether translation coherence problems can be solved with better software. However, we believe that practical software solutions can only partially solve the problem of flushing all translation structures and cannot easily solve the problem of identifying all vCPUs as translation coherence targets. Consider the problem of flushing translation structures. One might consider tackling this by modifying the guest-hypervisor interface to enable the hypervisor to use existing ISA support (e.g., invlpg) to selectively invalidate TLB entries. But this only fixes TLB invalidation – no architectures currently support selective invalidation instructions for MMU caches and nTLBs, so these would still have to be flushed.

Even if this problem could be solved, making target-side translation coherence handling lightweight is challenging. Fundamentally, handling translation coherence in software means that CPU context switches are unavoidable. One alternative to VM exits might be to switch to lighter-weight interrupts to query the guest OS for GVP-SPP mappings. Unfortunately, even these interrupts remain expensive. We profiled interrupt costs using microbenchmarks on Intel's Haswell machines and found that they require 640 cycles on average, which is just half of the average of 1300 cycles required for a VM exit. Contrast this with HATRIC, which entirely eliminates these costs by *never* disrupting the operation of the guest OS or requiring context switching.

4 HARDWARE DESIGN

We now detail HATRIC's design. HATRIC achieves all three goals from Section 1. It does so by adding co-tags to translation structures. This obviates the need for full translation structure flushes by more precisely identifying invalidation targets. HATRIC then exposes these co-tags to the cache coherence protocol to precisely identify coherence targets and to eliminate VM exits.

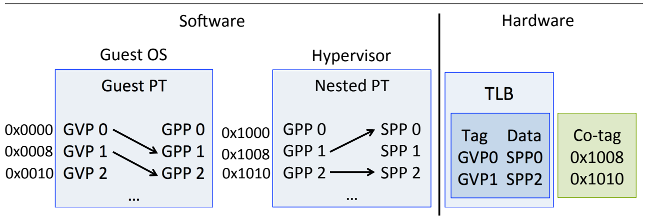

Figure 4: We add co-tags to store the system physical addresses where nested page table entries are stored. In our final implementation, we only store a subset of the system physical address bits.

4.1 Co-Tags

What are co-tags? Consider the page tables of Figure 4 and suppose that the hypervisor modifies the GPP 2–SPP 2 nested page table mapping, making the TLB entry caching information about SPP 2 stale. Since the TLB caches GVP–SPP mappings rather than GPP–SPP mappings, this means that we would like to selectively invalidate GVP 1–SPP 2 from the TLB, and although not shown, corresponding MMU cache and nTLB entries. Co-tags allow us to do this by logically acting as tag extensions that allow precise identification of translations when the hypervisor does not know the GVP. In other words, each TLB, MMU cache, and nTLB entry has its own co-tag. Co-tags store the system physical address of the nested page table entry (nL1 from the bottom-most row in Figure 1). For example, GVP 1–SPP 2 uses the nested page table entry at system physical address 0x1010, which is stored in the co-tag.

What do co-tags accomplish? Co-tags not only permit more precise translation identification, but can also be piggybacked on existing cache coherence protocols. When the hypervisor modifies a nested page table translation, cache coherence protocols detect the modification to the system physical address of the page table entry. Ordinarily, all private caches respond so that only one amongst them holds the up-to-date copy of the cache line storing the nested page table entry. With co-tags, HATRIC extends cache coherence as follows. Coherence messages, previously restricted to just private caches, are now also relayed to translation structures too. Co-tags are used to identify which (if any) TLB, MMU cache, and nTLB entries correspond to the modified nested page table cache line. Overall, this means that co-tags: ⓐ pick up on nested page table changes entirely in hardware, without the need for IPIs, VM exits, or invlpg instructions; ⓑ rely on, without fundamentally changing, existing cache coherence protocols; ⓒ permit selective TLBs, MMU caches, and nTLBs rather than flushes.

How are co-tags implemented? Logically, co-tags act as tag extensions. Physically, we implement co-tags in separate set-associative structures, one per translation structure. Each translation structure's co-tag array maintains one co-tag per translation structure entry. It is possible to use either set-associative CAM or SRAM structures to realize co-tag arrays. Naturally, CAM-based lookups are quicker. We therefore focus on set-associative CAM structures, similar to the reverse CAM structures used for UNITD [57].

Thus far, we have assumed that co-tags store all the bits associated with the physical address of its corresponding page table entry. This, however, is a naive implementation with an important drawback.

Like the reverse CAMs used in UNITD, co-tags storing all 64 physical address bits become as large as TLB entries themselves. Even worse, co-tags triple the area needed for MMU cache and nTLB entries. Since address translation can account for 13-15% of processor energy [21, 30, 31, 33, 60], using such large CAMs implies unacceptable area and energy overheads.

Therefore, our HATRIC implementation uses co-tags with a fewer number of bits than the 64 physical address bits. This decreased resolution means that groups of TLB entries, rather than individual TLB entries, may be invalidated when one nested page table entry is changed. However, judiciously-sized co-tags achieve a good balance between invalidation precision, and area/energy overheads. Section 6 shows, using detailed RTL modeling, that 2-byte co-tags (a per-core area overhead of 0.2%) strike a good balance.

Who sets co-tags? For performance, co-tags must be set by hardware without an OS or hypervisor interrupt. HATRIC uses the page table walker to do this. On TLB, MMU cache, and nTLB misses, the page table walker performs a two-dimensional page table walk. In so doing, it infers the system physical address of the page table entries and stores it in the TLB, MMU cache, and nTLB co-tags.

4.2 Coherence States and Initiators

Since TLBs, MMU caches, and nested TLBs are read-only structures, HATRIC integrates them into existing cache coherence protocol in a manner similar to read-only instruction caches. We describe HATRIC's operation on a directory-based MESI protocol, with the coherence directories located at the shared LLC cache banks and use dual-grain coherence directories from recent work [69].

Translation structure coherence states: Since translation structures are read-only, their entries require only two states: Shared (S), and Invalid (I). These two states may be realized using valid bits. When a translation is entered into the TLB, MMU cache, or nTLB, the valid bit is set, representing the S state. At this point, the translation can be accessed by the local CPU. The translation structure entry remains in this state until it receives a coherence message. Co-tags are compared to incoming messages. When an invalidation request matches the co-tag, the translation entry is invalidated.

Translation coherence initiators: With HATRIC, translation coherence activity is initated by either the hardware page table walker or privileged software (i.e., the OS or hypervisor). Page table walkers are hardware finite state machines that are invoked on TLB misses. They traverse page tables and are responsible for filling translation information into the translation structures and setting the co-tags. Page table walkers cannot map or unmap pages. On the other hand, the OSes and hypervisor can map and unmap page table entries using standard load/store instructions. HATRIC picks up these changes and keeps private cache and translation structures coherent.

4.3 Coherence Directory and Co-Tag Interaction

HATRIC requires some changes to the coherence directory. We discuss these changes and their design implications in this section.

Tracking translation entries: There are several ways to implement coherence directories, but we assume banked directories placed in conjunction with the LLC (one bank per LLC bank). HATRIC's coherence directories track, as is usual, both cache lines that store non-page table data and page table data. However, some changes are necessary to interpret the directory entries maintaining page table data. Specifically, all cache coherence protocol directory entries maintain a sharer list that indicates which CPUs have their private caches hosting each cache block. Directory entries for non-page table data can be left unchanged. However, we need to change how the sharer list is updated and interpreted for directory entries tracking page table data. Consider a directory entry for page table data with a sharer list of {CPU 0, CPU 1, CPU 3}. Ordinarily, this sharer list means that a cache line storing page table data is available in the private caches of CPU 0, CPU 1, and CPU 3. In other words, this sharer list tells us nothing about which translation structures (i.e., TLBs, MMU caches, and nTLBs) maintain page table entries from this page table cache line. Instead, HATRIC updates and interprets these sharer lists different. That is, a sharer list with CPU 0, CPU 1, and CPU 3 indicates that these CPUs may be caching page table entries from the corresponding page table cache line in any of the private caches or translation structures.

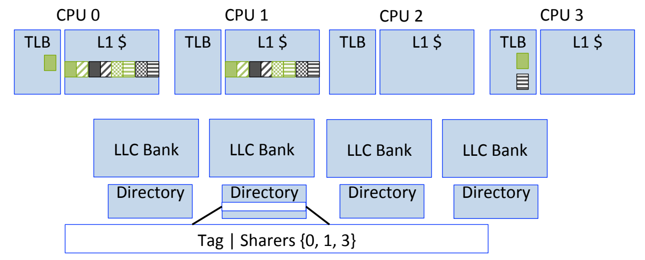

Figure 5: Coherence directories identify translation structures caching page table entries, aside from private L1 cache contents.

Figure 5 shows an example of how sharer lists are maintained. We show a system with four CPUs, with an LLC and adjoining directory, each banked four ways. CPU 0 maintains a cache line of eight page table entries, one of which also exists in its TLB. Meanwhile CPU 1 also maintains this cache line, but does not cache any of the page table entries in its TLB, while CPU 3 caches one of the page table entries in the TLB but none in the private cache. The sharer list in the directory entry does not differentiate among these cases however, and merely tracks the fact that these CPUs maintain a private copy of at least one of the page table entries in the cache line in one of the private caches or translation structures (i.e., the TLB or, although not shown, the MMU caches or nTLBs).

It is possible to modify the sharer list to provide more specific information about where translations reside. However, this requires additional bits of storage in directory entries. Instead, we choose this *pseudo-specific* implementation to simplify hardware. Naturally, this may result in spurious coherence messages – when a CPU modifies page table contents and invalidation messages need to be sent to the sharers, they are relayed to the L1 caches *and* all translation structures regardless of which ones actually cache page tables. In Figure 5, for example, this results in spurious coherence activity to CPU 3's L1 cache. In practice, because modifications of the page table are rare compared to other coherence activity, this additional traffic is tolerable. Ultimately, the gains from eliminating high-latency software TLB coherence far outweigh these relatively minor overheads (see Section 6).

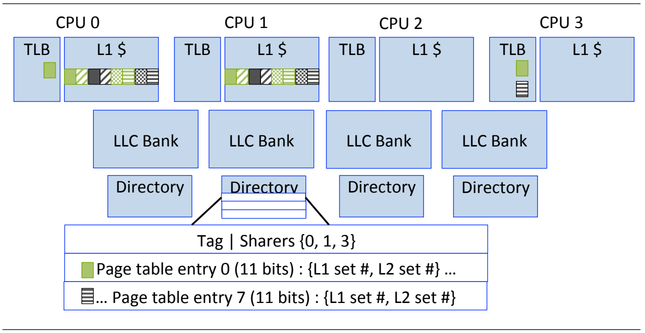
CPU 0
CPU 1
CPU 2
CPU 3
TLB
L1 $
LLC Bank
Directory
Tag | Sharers {0, 1, 3}
Page table entry 0 (11 bits) : {L1 set #, L2 set #} ...
... Page table entry 7 (11 bits) : {L1 set #, L2 set #}

Figure 6: We use additional coherence directory entries adjacent to the directory entry tracking sharers, to track TLB set numbers.

Coherence granularity: HATRIC's directory entries store information at the cache line granularity. Since x86-64 systems cache 8 page table entries per 64-byte cache line, similar to false sharing in caches, HATRIC conservatively invalidates all translation structure entries caching these 8 page table entries if a single page table entry is modified. Consider CPU 3 in Figure 5, where the TLB caches two translations mapped to the same cache line. If any CPU modifies either one of these translations, HATRIC has to invalidate both TLB entries. This has implications on the size of co-tags. Recall that in Section 4.1, we stated that co-tags use a subset of the address bits. We want to use the least significant and hence highest entropy bits as co-tags. But since cache coherence protocols track groups of 8 translations, co-tags do not store the 3 least significant address bits. Our 2 byte co-tags use bits 18-3 of the system physical address storing the page table. Naturally, this means that translations from different addresses in the page table may alias to the same co-tag. In practice, we find this has little adverse effect on HATRIC's performance.

Looking up co-tags: Thus far, we have ignored details of how the translation structure co-tags are looked up. However, when directory entries identify sharers, it is important that the TLB, MMU cache, and nTLB lookup and invalidation messages they relay be energy efficient. As we have already detailed, we achieve better energy efficiency than prior work on UNITD by architecting the co-tags as set-associative CAMs. This begs the following question: how can directory entries identify the target set number in the various translation structures? We use separate approaches for TLBs and MMU caches/nTLBs.

① TLBs: HATRIC uses the simple approach for L1 and L2 TLBs and records set numbers in the directory. Since a directory entry essentially tracks information about all 8 page table entries within a cache line, we need to record L1 and L2 TLB set numbers for each individual page table entry. Modern systems (e.g., Intel's Broadwell or Skylake architectures) tend to use 64 entry L1 TLBs and 512-1536 entry L2 TLBs that are 4 way and 8-12 way associative, respectively. Therefore, L1 and L2 TLBs tend to use up to 16 and 128 sets respectively, meaning that they need 4 and 7 bits for set identification. This amounts to a total of 88 bits to store all the L1 and L2 TLB set numbers for all 8 page table entries in a cache line.

Consequently, we studied two options for embedding set numbers in the directory. In one option, we use an additional coherence directory entry to record TLB set information. We save storage space

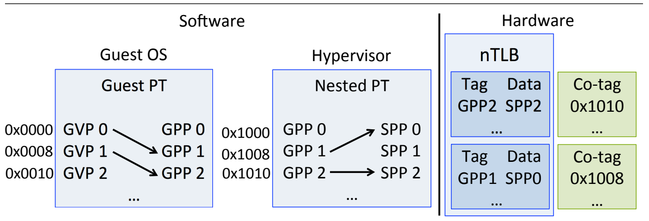

Figure 7: MMU cache and nTLB set numbers can be inferred directly from the physical address of the page table entry being modified, so there is no need to store them in the coherence directory entries.

by using 6 bits to record L1 and L2 TLB set numbers for each of the 8 page table entries. This results in a usage of 48 bits, matching the size of coherence directory entries. The tradeoff is that this approach requires lookup of multiple TLB sets to find the matching co-tag, expending more energy. In the second option (shown in Figure 6), we use two additional directory coherence directory entries, which comfortably maintain all 11 set identification bits per page table entry. In both scenarios, these additional directory entries are managed such that their allocation and replacement are performed in tandem with the original directory entry storing sharer information. We have modeled both options and have found no performance difference between the two approaches and only a minor energy difference. We assume the second approach for the remainder of this work since we have found it to be more energy efficient.

② MMU caches and nTLBs: One might initially consider treating MMU caches and nTLBs in a manner that parallels TLBs and embed their set numbers in the directory too. However, we use a alternative storage-efficient approach. We observe that MMU caches and nTLBs cache information from a *single dimension* of the page tables, as opposed to TLBs, which cache information across both dimensions. This enables an implementation trick that precludes the need to embed MMU cache and nTLB set information in the directory.

Figure 7 shows this approach. We show the contents of a guest page table, and a nested page table. Furthermore, we focus on changes to the nTLB, with changes to MMU caches proceeding in a similar manner. Suppose that the nested page table entry mapping GPP 2 to SPP 2 is changed by a CPU. The coherence directory entry must consequently infer the set numbers within the MMU caches and nTLBs where this translation resides, so as to relay invalidation messages to them. We observe the following. The coherence protocol already tracks the physical address of the nested page table entry that is being changed ((0x1010) in our example). Since each page table entry is 8 bytes, the last 3 bits can be ignored. However, bits 11-3 of the physical address identifies which of the 512 page table entries in the nested page table page is being modified. It so happens that these 9 bits correspond exactly with 9 bits from the GPP. For example, if we're updating an L1 page table entry, bits 11-3 of the nested page table entry's physical address are equivalent to bits 20-12 of the GPP. Therefore, if – and this is true for all commercial MMU caches and nTLBs today – the MMU caches and nTLBs have fewer than 2^9 or 512 sets, the desired MMU cache/nTLB set can be uniquely identified by bits 11-3 of the physical address of the nested page table being changed. Since modern MMU caches and nTLBs use 2-8 sets (see Figure 7) today, there is no need to embed MMU cache or nTLB set numbers in the coherence directory entries.

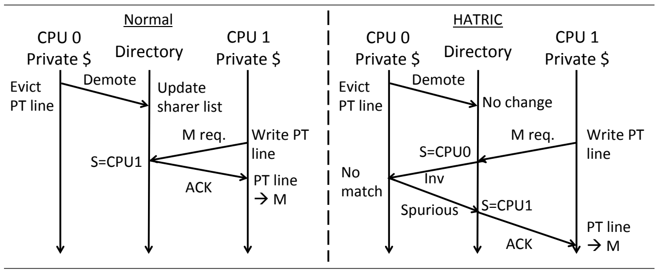

Figure 8: Coherence activity from the eviction of a cache line holding page table entries from CPU0's private cache. HATRIC **updates sharer list information lazily in response to cache line evictions.**

Co-tag lookup filtering: Naturally, we would like to initiate coherence activities for translation structures only when page tables are modified, to save co-tag lookup energy and reduce coherence traffic. Therefore, we need a way to distinguish directory entries corresponding to cache lines from page tables from those that store non-page table data. We achieve this by adding a single bit, a nested page table or nPT bit, for every coherence directory entry. The nPT bit is set by the hardware page table walker when any page table entry from the corresponding cache line is brought into the translation structures.

Silent versus non-silent evictions: Directories track translations in a coarse-grained and pseudo-specific manner. This has implications on cache line evictions. Usually, when a private cache line is evicted, the directory is sent a message to update the line's sharer list [69]. An up-to-date sharer list eliminates spurious coherence traffic. We continue to employ this strategy for non-page table cache lines but use a slightly different approach for page tables. When a cache line holding page table entries is evicted, its content may still be cached in the TLB, MMU cache, or nTLB. Even worse, other translations with matching co-tags may still be residing in the translation structures. One option is to detect all translations with matching co-tags and invalidate them. This hurts energy because of additional translation structure lookups, and hurts performance because of unnecessary TLB, MMU cache, and nTLB entry invalidations.

Figure 8 shows how HATRIC handles this problem, contrasting it with traditional cache coherence. Our approach is to essentially employ a slightly modified version of the well-known concept of silent evictions already used to reduce coherence traffic [61]. To showcase this in detail, suppose CPU 0 evicts a cache line containing page table entries. Both approaches relay a message to the coherence directory. Ordinarily, we remove CPU 0 from the sharer list. However, if HATRIC sees that this message corresponds to a cache line storing a page table (by checking the directory entry's page table bits), the sharer list is untouched. This means that if CPU 1 subsequently writes to the same cache line, HATRIC sends spurious invalidate messages to CPU 0, unlike traditional cache coherence. However, we mitigate frequency of spurious messages; when CPU 0 sees spurious coherence traffic, it sends a message back to the directory to demote CPU 0 from the sharer list. Sharer lists are hence lazily updated. Similarly, evictions from translation structures lazily update coherence directory sharer lists.

Directory evictions: Past work shows that coherence directory entry evictions require back-invalidations of the associated cache lines in the cores [69]. This is necessary for correctness; all lines in private caches must always have a directory entry. HATRIC extends this approach to relay back-invalidations to all translation structures.

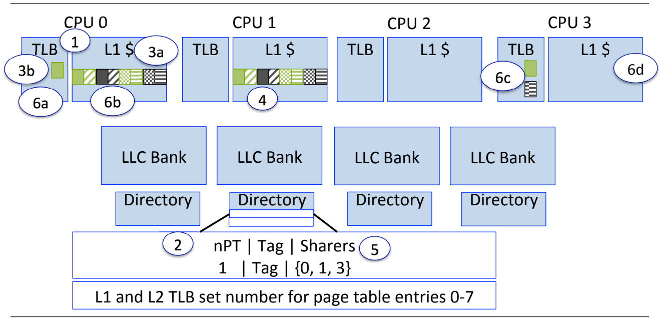

Figure 9: Coherence directories identify translation structures caching page table entries, aside from private L1 cache contents.

4.4 Putting It All Together

Figure 9 details HATRIC's overall operation. Initially, CPU 0's TLB and L1 caches are empty. On a memory access, CPU 0 misses in the TLB ①. Whenever a request is satisfied from a page table line in the L1 cache in the M, E, or S state, there is no need to initiate coherence transactions. However, suppose that the last memory reference in the page table walk from Figure 1 is absent in the L1 cache. A read request is sent to the coherence directory in step ②.

Two scenarios are possible. In the first, the translation may be uncached in the private caches and there is no coherence directory entry. A directory entry is allocated and the nPT bit is set. In the second scenario (shown in Figure 9) the request matches an existing directory entry. The nPT bit already is set and HATRIC reads the sharer list which identifies CPUs 1 and 3 as also caching the desired translation (and the 7 adjacent translations in the cache line) in shared state. In response, the cache line with the desired translations is sent back to CPU 0 (from CPU 1, 3, or memory, whichever is fastest), updating the L1 cache (3a) and TLB (3b). Subsequently, the sharer list adds CPU 0.

Now suppose that CPU 1 runs the hypervisor and unmaps the solid green translation from the nested page table in step ④. To transition the L1 cache line into the M state, the cache coherence protocol relays a message to the coherence directory. The corresponding directory entry is identified in ⑤ and we find that CPU 0 and 3 need to be sent invalidation requests. However, the sharer list is (i) coarse-grained and (ii) pseudo-specific. Because of (i), CPU 0 has to invalidate not only its TLB entry (6a) but also 8 translations in the L1 cache (6b) and CPU 3 has to invalidate the 2 TLB entries with matching co-tags (6c). Because of (ii), CPU 1's L1 cache receives a spurious invalidation message (6d).

4.5 Other Key Observations

Translation structure lookup latency and energy: We have modeled the area, latency, and energy implications of HATRIC on translation structures using CACTI. Our improvements result in a 0.2% area increase for each CPU, primarily from implementing co-tags. Despite this, translation structure accesses initiated by the local CPU see no change in access times and energy. This is because co-tags are not accessed on CPU-side lookups and are only used on translation coherence lookups. Further, when CPU-side lookups occur at the same time as coherence lookups, we prioritize the former. Finally, since HATRIC does not need associative lookups when probing co-tags, translation coherence lookups suffer little energy.

Other co-tag sizing issues: An important design issue is the relationship between translation structure size and co-tag resolution. In general, we need more co-tag bits for larger translation structures to ensure that false-positive matches do not become excessive. However, since we assume a set-associative co-tag implementation, the number of false-positives is restricted to the number of co-tags in a set, in the worst case. So unless translations become far more set-associative (an unlikely event since L2 TLBs already employ 12-way set associativity), false-positives are unlikely to become problematic.

Metadata updates: Beyond software changes to the translations, they may also be changed by hardware page table walkers. Specifically, page table walkers update dirty and access bits to aid page replacement policies [53]. However because these updates are picked up by the standard cache coherence protocol, HATRIC naturally handles these updates too.

Prefetching optimizations: Beyond simply invalidating stale translation structure entries, HATRIC could potentially update (or prefetch) the updated mappings into the translation structures. Since a thorough treatment of these studies requires an understanding of how to manage translation access bits while speculatively prefetching into translation structures [41], we leave this for future work.

Coherence protocols: We have studied a MESI directory based coherence protocol but we have also implemented HATRIC atop MOESI protocols too. HATRIC requires no fundamental changes to support these protocols.

Synonyms and superpages: HATRIC naturally handles synonyms or virtual address aliases. This is because synonyms are defined by unique translations in separate page table locations, and hence separate system physical addresses. Therefore, changing or removing a translation has no impact on other translations in the synonym set, allowing HATRIC to be agnostic to synonyms. Similarly, HATRIC supports superpages, which also occupy unique translation entries and can be easily detected by co-tags.

Multiprogrammed workloads: One might expect that when an application's physical page is remapped, there is no need for translation coherence activities to extend to the other applications, because they operate on distinct address spaces. Unfortunately, hypervisors do not know which physical CPUs an application executed on; all they know is the vCPUs the entire VM uses. Therefore, the hypervisor conservatively flushes even the translation structures of CPUs that never ran the offending application. HATRIC eliminates this problem by precisely tracking the correspondence between translations and CPUs.

Comparison to past approaches: HATRIC is inspired by UNITD [57]. HATRIC uses energy-frugal co-tags instead of UNITD's large reverse-lookup CAMs, achieving greater energy efficiency. We showcase this in Section 6 where we compare the efficiency of HATRIC versus an enhanced UNITD design for virtualization. Further, HATRIC extends translation coherence to MMU caches and nTLBs. Beyond UNITD, past work on DiDi [65] also targets translation coherence for non-virtualized systems. Similarly, recent work investigates translation coherence overheads in the context of die-stacked DRAM [46]. While this work mitigates translation coherence overheads, it does so specifically for non-virtualized x86 architectures. Finally, recent work uses software mechanisms to reduce translation overheads for guest page table modifications [47], while HATRIC tackles nested page table coherence.

5 METHODOLOGY

Our experimental methodology has two primary components. First, we modify KVM to implement paging on a two-level memory with die-stacked DRAM. Second, we use detailed cycle-accurate simulation to assess performance and energy.

5.1 Die-Stacked DRAM Simulation

We evaluate HATRIC's performance on a cycle-accurate simulation framework that models the operation of a 32-CPU Haswell processor. We assume 2GB of die-stacked DRAM with 4× the bandwidth of slower 8GB off-chip DRAM, similar to prior work [46]. Each CPU maintains 32KB L1 caches, 256KB L2 caches, 64-entry L1 TLBs, 512-entry L2 TLBs, 32-entry nTLBs [9], and 48-entry paging structure MMU caches [10]. Further, we assume a 20MB LLC. We model the energy usage of this system using the CACTI framework [44]. We use Ubuntu 15.10 Linux as our guest OS and evaluate HATRIC in detail using KVM and Xen.

We use a trace-based approach to drive our simulation framework. We collect instruction traces from our modified hypervisors with 50 billion memory references using a modified version of Pin [39] which tracks all GVPs, GPPs, and SPPs, as well as changes to the guest and nested page tables. In order to collect accurate paging activity, we collect these traces on a real system. Ideally, we would like this system to use die-stacked DRAM but since this technology is in its infancy, we are inspired by recent work [46] to modify a real-system to mimic the activity of die-stacking. We take an existing multi-socket NUMA platform and by introducing contention, creates two different speeds of DRAM. We use a 2-socket Intel Xeon E5-2450 system, running our software stack. We dedicate the first socket for execution of the software stack and mimicry of fast or die-stacked DRAM. The second socket mimics the slow or off-chip DRAM. It does so by running several instances of memhog on its cores. Similar to prior work [51, 53], we use memhog to carefully generate memory contention to achieve the desired bandwidth differential between the fast and slow DRAM of 4×. By using Pin to track KVM and Linux paging code on this infrastructure, we accurately generate instruction traces to test HATRIC.

5.2 KVM Paging Policies

Our goal is to showcase the overheads imposed by translation coherence on paging decisions rather than design the optimal paging policy. Thus, we pick well-known paging policies that cover a wide range of design options. For example, we have studied FIFO and LRU replacement policies, finding the latter to perform better as expected. We implement LRU policies in KVM by repurposing Linux's well-known pseudo-LRU CLOCK policy [19]. LRU alone doesn't always provide good performance since it is expensive to traverse page lists to identify good candidates for eviction from die-stacked

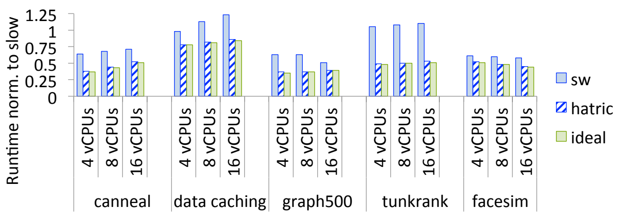

Figure 10: **For varying vCPUs, runtime of the best KVM paging policy without** HATRIC (sw)**, with** HATRIC (hatric)**, and with zero-overhead translation coherence** (ideal)**. All results are normalized to the case without die-stacked DRAM.**

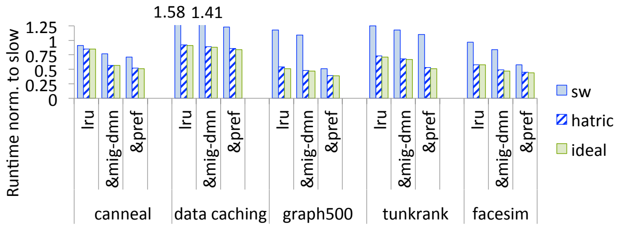

Figure 11: HATRIC's **performance benefits for KVM paging policies, with LRU, migration daemons** (mig-dmn)**, and prefetching** (pref.)**. Results are normalized to the case without die-stacked DRAM.**

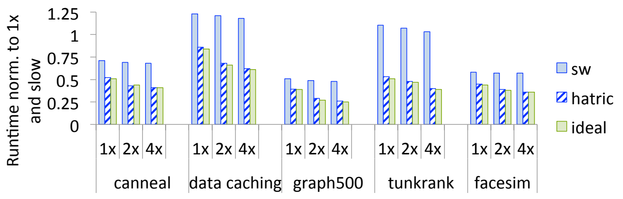

Figure 12: HATRIC's **performance benefits as a function of translation structure size.** 1× **indicates default sizes,** 2× **doubles sizes, and so on. All results are normalized to the case without die-stacked DRAM.**

memory. Instead, performance is improved by moving this operation off the critical path of execution; we therefor pre-emptively evict pages from die-stacked memory so that a pool of free pages are always maintained. We call this operation a *migration daemon* and combine it with LRU replacement. We have also investigated the benefits of page prefetching; that is, when an application demand fetches a page from off-chip to die-stacked memory, we also prefetch a set number of adjacent pages. Generally, we have found that the best paging policy uses a combination of these approaches.

5.3 Workloads

We focus on two sets of workloads. The first set comprises applications that benefit from the higher bandwidth of die-stacked memory. We use canneal and facesim from PARSEC [13], data caching and tunkrank from Cloudsuite [22], and graph500 as part of this group. We also create 80 multiprogrammed combinations of workloads from all the SPEC applications [28] to showcase the problem of imprecise target identification in virtualized translation coherence.

Our second group of workloads is made up of smaller-footprint applications whose data fits within the die-stacked DRAM. We use these workloads to evaluate HATRIC's overheads in situations where hypervisor-mediated paging (and hence translation coherence) between die-stacked and off-chip DRAM is rarer. We use the remaining PARSEC applications [13] and SPEC applications [28] for these studies.

6 EVALUATION

Performance as a function of vCPU counts: Figure 10 shows HATRIC's runtime, normalized as a fraction of application runtime in the absence of any die-stacked memory (no-hbm from Figure 2). We compare runtimes for the best KVM paging policies (sw), HATRIC, and ideal unachievable zero-overhead translation coherence (ideal). Further, we vary the number of vCPUs per VM and observe the following. HATRIC is *always* within 2-4% of the ideal performance. In some cases, HATRIC is instrumental in achieving any gains from die-stacked memory at all. Consider data caching, which slows down when using die-stacked memory because of translation coherence overheads, HATRIC cuts runtimes down to roughly 75% of the baseline runtime in all cases.

Figure 10 also shows that HATRIC is valuable across all vCPU counts. In some cases, more vCPUs exacerbate translation coherence overheads. This is because IPI broadcasts become more expensive and more vCPUs suffer VM exits. This is why data caching and tunkrank become slower (see sw) when vCPUs increase from 4 to 8. HATRIC eliminates these problems, flattening runtime improvements across all vCPU counts. In other scenarios, fewer vCPUs worsen performance since each vCPU performs more of the application's total work. Here, the impact of a full TLB, nTLB, and MMU cache flush for every page remapping is expensive (e.g. graph500 and facesim). HATRIC eliminates these overheads almost entirely.

Performance as a function of paging policy: Figure 11 also shows HATRIC performance but as a function of different KVM paging policies. We study three policies with 16 vCPUs. First, we show lru, which determines which pages to evict from die-stacked DRAM. We then add the migration daemon (&mig-dmn), and page prefetching (&pref).

Figure 11 shows HATRIC improves runtime substantially for any paging policy. Performance is best when all techniques are combined but HATRIC achieves 10-30% performance improvements even for just lru. Furthermore, Figure 11 shows that translation coherence overheads can often be so high that the paging policy itself makes little difference to performance. Consider tunkrank, where the difference between lru versus the &pref bars is barely 2-3%. With HATRIC, however, paging optimizations like prefetching and migration daemons help.

Impact of translation structure sizes: One of HATRIC's advantages is that it converts translation structure flushes to selective invalidations. This improves TLB, MMU cache, and nTLB hit rates substantially, obviating the need for expensive two-dimensional page table walks. We expect HATRIC to improve performance even more as translation structures become bigger (and flushes needlessly evict more entries). Figure 12 quantifies the relationship. We vary TLB,

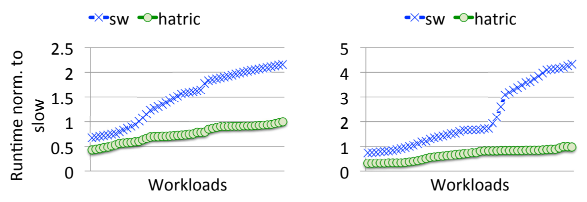

Figure 13: (Left) Weighted runtime for all 80 multiprogrammed workloads on VMs without (sw) **and with** HATRIC (hatric)**; (Right) the same for the slowest application in mix.**

nTLB, and MMU cache sizes from the default (see Section 5) to double (2×) and quadruple (4×) the number of entries.

Figure 12 shows that translation structure flushes largely counteract the benefits of greater size. Specifically, the sw results see very small improvements, even when sizes are quadrupled. Inter-DRAM page migrations essentially flush the translation structures so often that additional entries are not effectively leveraged. Figure 12 shows that this is a wasted opportunity since zero-overhead translation coherence (ideal) actually does enjoy 5-7% performance benefits. HATRIC solves this problem, comprehensively achieving within 1% of the ideal, thereby exploiting larger translation structures.

Multiprogrammed workloads: We now focus on multiprogrammed workloads made up of sequential applications. Each workload runs 16 Spec benchmarks on a Linux VM atop KVM. As is standard for multiprogrammed workloads, we use two performance metrics [62]. The first is weighted runtime improvement, which captures overall system performance. The second is the runtime improvement of the slowest application in the workload, capturing fairness.

Figure 13 shows our results. The graph on the left plots the weighted runtime improvement, normalized to cases without die-stacked DRAM. As usual, sw represents the best KVM paging policy. The x-axis represents the workloads, arranged in ascending order of runtime. The lower the runtime, the better the performance. Similarly, the graph on the right of Figure 12 shows the runtime of the slowest application in the workload mix; again, lower runtimes indicate a speedup in the slowest application.

Figure 13 shows that translation coherence can be disastrous to the performance of multiprogrammed workloads. More than 70% of the workload combinations suffer performance degradation when using die-stacked memory without HATRIC. These applications suffer from unnecessary translation structure flushes and VM exits, caused by software translation coherence's imprecise target identification. The runtime of 11 workloads is more than double. Additionally, translation coherence degrades application fairness. For example, in more than half the workloads, the slowest application's runtime is 2×+ with a maximum of 4×+. Applications that struggle are usually those with limited memory-level parallelism that benefit little from the higher bandwidth of die-stacked memory and instead, suffer from the additional translation coherence overheads.

HATRIC solves these issues, achieving improvements for every single weighted runtime and even for the slowest applications. In fact, HATRIC eliminates translation coherence overheads, reducing runtime to 50-80% of the baseline without die-stacked DRAM. The key enabler is HATRIC's precise identification of coherence targets; applications that do not need to participate in translation coherence operations have their translation structure contents left unflushed and do not suffer VM exits.

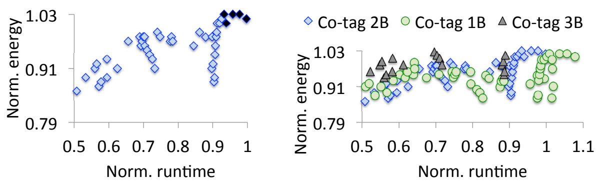

Figure 14: (Left) Performance-energy plots for default HATRIC **configuration compared to a baseline with the best paging policy; and (Right) impact of co-tag size on performance-energy tradeoffs.**

Performance-energy tradeoffs: Intuitively, we expect that since HATRIC reduces runtime substantially, it should reduce static energy sufficiently to offset the higher energy consumption from the introduction of co-tags. Indeed, this is true for workloads that have sufficiently large memory footprints to trigger inter-memory paging. However, we also assess HATRIC's energy implications on workloads that do not frequently remap pages (i.e. their memory footprints fit comfortably within die-stacked DRAM).

The graph on the left of Figure 14 plots all the workloads including the single-threaded and multithreaded ones that benefit from die-stacking and those whose memory needs fit entirely in die-stacked DRAM. The x-axis plots the workload runtime, as a fraction of the runtime of sw results. The y-axis plots energy, similarly normalized. We desire points that lie on the lower-left corner of the graph.

Figure 14 shows that HATRIC always boosts performance, and almost always improves energy too. Energy savings of 1-10% are routine. In fact, HATRIC even improves the performance and energy of many workloads that do not page between the two memory levels significantly. This is because these workloads still remap pages to defragment memory (to support superpages) and HATRIC mitigates the associated translation coherence overheads. There are some rare instances (highlighted in black) where energy does exceed the baseline by 1-1.5%. These are workloads for whom efficient translation coherence does not make up for the additional energy of the co-tags. Nevertheless, these overheads are low, and their instances rare.

Co-tag sizing: We now turn to co-tag sizing. Excessively large co-tags consume significant lookup and static energy, while small ones force HATRIC to invalidate too many translation structures on a page remap. The graph on the right of Figure 14 shows the performance-energy implications of varying co-tag size from 1 to 3 bytes.

First and foremost, 2B co-tags, our design choice, provides the best balance of performance and energy. While 3B co-tags track page table entries at a finer granularity, they only modestly improve performance over 2B co-tags, but consume much more energy. Meanwhile 1B co-tags suffer in terms of both performance and energy. Since 1B co-tags have a coarser tracking granularity, they invalidate more translation entries from TLBs, MMU caches, and nTLBs than larger co-tags. While the smaller co-tags do consume less lookup and static energy, these additional invalidations lead to more expensive two-dimensional page table walks and a longer system runtime. The end result is an increase in energy.

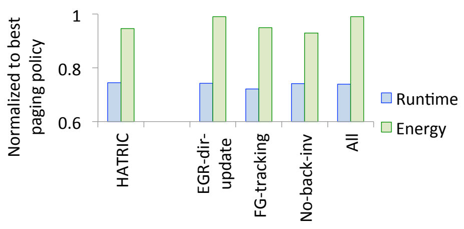

Figure 15: Baseline HATRIC **versus approaches with eager update of directory on cache and translation structure evictions** (EGR-dir-update)**, fine-grained tracking of translations** (FG-tracking)**, and an infinite directory with no back-invalidations** (No-back-inv)**.** All **combines these approach. We show average runtime and energy, normalized to the metrics for the best paging policy without** HATRIC**.**

Coherence directory design decisions: Section 4 detailed the nuances modifying traditional coherence directories to support translation coherence. Figure 15 captures the performance and energy (normalized to those of the best paging policy or sw in previous graphs) of these approaches. We consider the following options, beyond baseline HATRIC:

EGR-dir-update: This is a design that eagerly updates coherence directories whenever a translation entry is evicted from a CPU's L1 cache or translation structures. While this does reduce spurious coherence messages, it requires expensive lookups in translation structures to ensure that entries with the same co-tag have been evicted. Figure 15 shows that the performance gains from reduced coherence traffic is almost negligible, while energy does increase, relative to HATRIC.

FG-tracking: We study a hypothetical design with greater specificity in translation tracking. That is, coherence directories are modified to track whether translations are cached in the TLBs, MMU caches, nTLBs, or L1 caches. Unlike HATRIC, if a translation is cached in only the MMU cache but not the TLB, the latter is not sent invalidation requests. Figure 15 shows that while one might expect this specificity to result in reduced coherence traffic, system energy is slightly higher than HATRIC. This is because more specificity requires more complex and area/energy intensive coherence directories. Further, since the runtime benefits are small, we believe HATRIC remains the smarter choice.

No-back-inv: We study an ideal design with infinitely-sized coherence directories which never need to relay back-invalidations to private caches or translation structures. We find that this does reduce energy and runtime, but not significantly from HATRIC.

All: Figure 15 compares HATRIC to an approach which marries all the optimizations discussed. HATRIC almost exactly meets the same performance and is more energy-efficient, largely because the eager updates of coherence directories add significant translation structure lookup energy.

Comparison with UNITD: We now compare HATRIC to prior work on UNITD [57]. To do this, we first upgrade the baseline UNITD design in several ways. First and most importantly, we extend virtualization support by storing the system physical addresses of nested

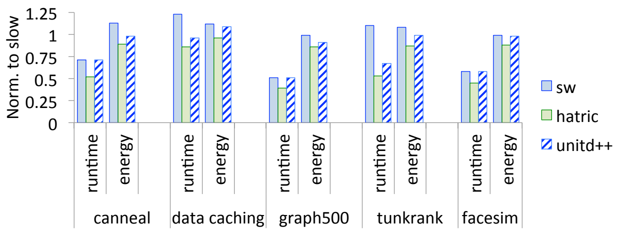

Figure 16: Comparison of HATRIC's **performance and energy versus** UNITD++**. All results are normalized to results for a system without die-stacked memory and compared to** sw**.**

page tables entries in the originally proposed reverse-lookup CAM [57]. Second, we extend UNITD to work seamlessly with coherence directories. We call this upgraded design UNITD++.

Figure 16 compares HATRIC and UNITD++ results, normalized to results from the case without die-stacked DRAM. As expected both approaches outperform a system with only traditional software-based translation coherence (sw). However, HATRIC provides an additional 5-10% performance boost versus UNITD++ by also extending the benefits of hardware translation coherence to MMU caches and nTLBs. Further, HATRIC is more energy efficient than UNITD++ as it boosts performance (saving static energy) but also does not need reverse-lookup CAMs.

Xen results: To assess HATRIC's generality, we have begun studying its effectiveness on Xen. Because our memory traces require months to collect, we have thus far evaluated canneal and data caching, assuming 16 vCPUs. Our initial results show that Xen's performance is improved by 21% and 33% for canneal and data caching respectively, over the best paging policy employing software translation.

7 CONCLUSION

We propose HATRIC, folding translation coherence atop existing hardware cache coherence protocols. We achieve this with simple modifications to translation structures (TLBs, MMU caches, and nTLBs) and with state-of-the-art coherence protocols. HATRIC is readily-implementable and beneficial for upcoming systems, especially as they rely on page migration to exploit heterogeneous memory systems.

8 ACKNOWLEDGMENTS

We thank Martha Kim and Ricardo Bianchini for helpful feedback on early drafts of this paper. We also thank Daniel Sanchez, David Nellans, and Jayneel Gandhi for suggestions on improving the final version. We thank the National Science Foundation, which partially supported this work through grants 1253700 and 1337147, as well as VMware for partial support of this work through a research gift.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1168857.1168860

[2] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2694344.2694381

[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)*. IEEE Computer Society, Washington, DC, USA, 476–487. http://dl.acm.org/citation.cfm?id=2337159.2337214

[4] Andrea Arcangeli. 2010. Transparent Hugepage Support. *KVM Forum* (August 2010). Retrieved April 18, 2017 from https://www.linux-kvm.org/images/9/9e/2010-forum-thp.pdf

[5] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 416–427. http://dl.acm.org/citation.cfm?id=2337159.2337207

[6] Amitabha Banerjee, Rishi Mehta, and Zach Shen. 2015. NUMA Aware I/O in Virtualized Systems. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI '15)*. IEEE Computer Society, Washington, DC, USA, 10–17. https://doi.org/10.1109/HOTI.2015.17

[7] 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)*. ACM, New York, NY, USA, 48–59. 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 *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 307–318. https://doi.org/10.1145/2000064.2000101

[9] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/1346281.1346286

[10] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[11] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 63–76. https://doi.org/10.1145/3037697.3037705

[12] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[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)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[14] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. 2006. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 469–479. https://doi.org/10.1109/MICRO.2006.18

[15] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R. Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. 2014. Improving DRAM performance by parallelizing refreshes with accesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 356–367. https://doi.org/10.1109/HPCA.2014.6835946

[16] Jonathan Corbet. 2016. Heterogeneous memory management. (2016). Retrieved April 18, 2017 from http://lwn.net/Articles/684916

[17] 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)*. ACM, New York, NY, USA, 435–448. https://doi.org/10.1145/3037697.3037704

[18] Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. 2013. A Circuit-architecture Co-optimization Framework for Exploring Nonvolatile Memory Hierarchies. *ACM Trans. Archit. Code Optim.* 10, 4, Article 23 (Dec. 2013), 22 pages. https://doi.org/10.1145/2541228.2541230

[19] Malcolm C. Easton and Peter A. Franaszek. 1979. Use Bit Scanning in Replacement Decisions. *IEEE Trans. Comput.* C-28, 2 (Feb 1979), 133–141. https://doi.org/10.1109/TC.1979.1675302

[20] Babak Falsafi, Tim Harris, Dushyanth Narayanan, and David A. Patterson. 2016. Rack-scale Computing (Dagstuhl Seminar 15421). *Dagstuhl Reports* 5, 10 (2016), 35–49. https://doi.org/10.4230/DagRep.5.10.35

[21] Dongrui Fan, Zhimin Tang, Hailin Huang, and Guang R. Gao. 2005. An Energy Efficient TLB Design Methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED '05)*. ACM, New York, NY, USA, 351–356. https://doi.org/10.1145/1077603.1077688

[22] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[23] 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)*. IEEE Computer Society, Washington, DC, USA, 178–189. 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 *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 707–718. https://doi.org/10.1109/ISCA.2016.67

[25] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 231–242. http://dl.acm.org/citation.cfm?id=2643634.2643659

[26] Jerome Glisse. 2016. HMM (Heterogeneous memory management) v5. (2016). Retrieved April 18, 2017 from http://lwn.net/Articles/619067

[27] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*. ACM, New York, NY, USA, 39–51. https://doi.org/10.1145/2731186.2731187

[28] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[29] Intel. 2015. Introducing Intel Optane Technology - Bringing 3D XPoint Memory to Storage and Memory Products. (2015). Retrieved April 18, 2017 from https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage\-and-memory-products

[30] Toni Juan, Tomas Lang, and Juan J. Navarro. 1997. Reducing TLB Power Requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design (ISLPED '97)*. ACM, New York, NY, USA, 196–201. https://doi.org/10.1145/263272.263332

[31] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. 2002. Generating Physical Addresses Directly for Saving Instruction TLB Energy. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 185–196. http://dl.acm.org/citation.cfm?id=774861.774882

[32] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel H. Loh. 2015. Enabling Interposer-based Disintegration of Multi-core Processors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 546–558. https://doi.org/10.1145/2830772.2830808

[33] Vasileios Karakostas, Jayneel Gandhi, Adrian Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643. https://doi.org/10.1109/HPCA.2016.7446100

[34] Anshuman Khandaul. 2016. Define coherent device memory node. (2016). Retrieved April 18, 2017 from http://lwn.net/Articles/404403

[35] Joonyoung Kim, Younsu Kim, undefined, undefined, undefined, and undefined. 2014. HBM: Memory solution for bandwidth-hungry processors. *2014 IEEE Hot Chips 26 Symposium (HCS)* 00 (2014), 1–24. https://doi.org/doi.ieeecomputersociety.org/10.1109/HOTCHIPS.2014.7478812

[36] 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 Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[37] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of*

the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15). USENIX Association, Berkeley, CA, USA, 277–289. http://dl.acm.org/citation.cfm?id=2813767.2813788

[38] Gabriel Loh and Mark D. Hill. 2012. Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap. *IEEE Micro* 32, 3 (May 2012), 70–78. https://doi.org/10.1109/MM.2012.25

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[40] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. 2013. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.* 10, 1, Article 2 (April 2013), 38 pages. https://doi.org/10.1145/2445572.2445574

[41] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 233–247. https://doi.org/10.1145/2872362.2872399

[42] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why On-chip Cache Coherence is Here to Stay. *Commun. ACM* 55, 7 (July 2012), 78–89. https://doi.org/10.1145/2209249.2209269

[43] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[44] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 3–14. https://doi.org/10.1109/MICRO.2007.33

[45] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 89–104. https://doi.org/10.1145/844128.844138

[46] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 188–200. https://doi.org/10.1109/PACT.2015.30

[47] Jiannan Ouyang, John R. Lange, and Haoqiang Zheng. 2016. Shoot4U: Using VMM Assists to Optimize TLB Operations on Preempted vCPUs. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '16)*. ACM, New York, NY, USA, 17–23. https://doi.org/10.1145/2892242.2892245

[48] J. T. Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477494

[49] Sujay Phadke and Satish Narayanasamy. 2011. MLP aware heterogeneous memory system. In *2011 Design, Automation Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2011.5763155

[50] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 558–567. https://doi.org/10.1109/HPCA.2014.6835964

[51] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 258–269. https://doi.org/10.1109/MICRO.2012.32

[52] Binh Pham, Jan Vesely, Gabriel Loh, and Abhishek Bhattacharjee. 2015. *Using TLB Speculation to Overcome Page Splintering in Virtual Machines*. Rutgers Technical Report DCS-TR-713. Department of Computer Science, Rutgers University, Pistcataway, NJ.

[53] 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 *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2830772.2830773

[54] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[55] Dulloor Subramanya Rao and Karsten Schwan. 2010. vNUMA-mgr: Managing VM memory on NUMA platforms. In *2010 International Conference on High Performance Computing*. 1–10. https://doi.org/10.1109/HIPC.2010.5713191

[56] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. 2013. Optimizing Virtual Machine Scheduling in NUMA Multicore Systems. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 306–317. https://doi.org/10.1109/HPCA.2013.6522328

[57] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. 2010. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416643

[58] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. 2015. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 79–91. https://doi.org/10.1145/2749469.2750379

[59] Agam Shah. 2014. Micron's Revolutionary Hybrid Memory Cube Tech is 15 Times Faster than Today's DRAM. (2014). Retrieved April 18, 2017 from http://www.pcworld.com/article/2366680/computer-memory-overhaul-due-with-microns-hmc-in-early-2015.html

[60] Avinash Sodani. 2011. Race to Exascale: Opportunities and Challenges. (2011). Retrieved April 18, 2017 from https://www.microarch.org/micro44/files/Micro%20Keynote%20Final%20-%20Avinash%20Sodani.pdf

[61] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.

[62] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. 2016. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (Oct 2016), 3071–3087. https://doi.org/10.1109/TPDS.2016.2526003

[63] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/195473.195531

[64] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. https://doi.org/10.1109/ISPASS.2016.7482091

[65] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 340–349. https://doi.org/10.1109/PACT.2011.65

[66] VMware. 2011. Performance Best Practices for VMware vSphere 5.0. (2011). Retrieved April 18, 2017 from https://www.vmware.com/pdf/Perf_Best_Practices_vSphere5.0.pdf

[67] Yuan Xie. 2011. Modeling, Architecture, and Applications for Emerging Memory Technologies. *IEEE Des. Test* 28, 1 (Jan. 2011), 44–51. https://doi.org/10.1109/MDT.2011.20

[68] Yuan Xie. 2013. *Emerging Memory Technologies: Design, Architecture, and Applications*. Springer Publishing Company, Incorporated.

[69] Jason Zebchuk, Babak Falsafi, and Andreas Moshovos. 2013. Multi-grain Coherence Directories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 359–370. https://doi.org/10.1145/2540708.2540739