VIRTUAL MEMORY FOR NEXT-GENERATION TIERED MEMORY ARCHITECTURES

By

ZI YAN

A dissertation submitted to the

Schoool of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Abhishek Bhattacharjee

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2019

ABSTRACT OF THE DISSERTATION

Virtual Memory for Next-Generation Tiered Memory Architectures

by Zi Yan

Dissertation Director:

Abhishek Bhattacharjee

Virtual memory offers a simple hardware abstraction to programmers freeing them from the tedious process of manual memory management. However, the emergence of new memory technologies is posing challenges for conventional virtual memory. Homogeneous memory systems are being replaced by complex heterogeneous systems with multiple memory devices with different latency, bandwidth, and capacity characteristics. This poses two problems. The first is that operating systems (OSes) must migrate pages among the heterogeneous memory devices based on attributes like page hotness and proximity to the compute unit/accelerator that uses the data. As this thesis shows, current support for page migration is infeasibly slow on emerging hardware, both due to the slow speeds of data movement and metadata update operations like TLB shootdowns. The second is that the ever-increasing aggregate capacities of these emerging heterogeneous memory systems pose immense pressure on TLBs, aggravating address translation overheads. **This thesis addresses these problems by proposing modest hardware/software techniques that achieve a more efficient virtual memory system via fast hardware support for translation coherence, software support for faster page copies, and hardware/software co-design that compresses TLB entries to reduce address translation overheads**.

Page migration is the means by which OSes can dynamically shift data to the memory devices that best benefit latency, bandwidth, capacity, and persistence characteristics in different phases of the program lifetime. The key to good performance is fast page migration. This thesis attacks two bottlenecks that currently constrain page migration performance — high-overhead translation coherence and low-throughput page copying. To mitigate the first source of overhead, this thesis implements hardware support for translation coherence by fusing it with existing cache coherence protocols. To mitigate the second source of overhead, this thesis implements OS support that parallelizes, aggregates, and consolidates page migration operations to maximize migration throughput.

Heterogeneous systems are also continuing a trend that has long been seen with traditional homogeneous memory systems — the drive towards ever-increasing memory capacities. Specific types of emerging systems with die-stacking technologies and byte-addressable persistent memories are further accelerating the total physical memory capacity that must be addressable for each process. Consequently, page tables are becoming bigger and TLB misses more frequent. To mitigate increasing address translation overheads, this thesis offers software techniques to facilitate the possibility of compressing TLB entries which rely on translation contiguity.

In summary, this work upgrades virtual memory to effectively support heterogeneous memory systems with high-performance page migration and scalable address translation. In so doing, this dissertation identifies bottlenecks in the existing virtual memory, profiles the performance impacts of these bottlenecks, and proposes hardware and software solutions to remedy them.

Acknowledgements

First of all, I would like to thank my advisor Abhishek Bhattacharjee. He provided enormous help throughout my Ph.D. study both personally and professionally. He always gave me the freedom to explore my own research and pushed me to improve in all aspect of my career. I will be always grateful for his mentorship.

I would also like to thank Daniel Lustig, David Nellans, and the rest of my thesis committee: Ulrich Kremer, Sudarsun Kannan, and Gabriel Loh. Their feedback and suggestions helped me greatly improve my thesis. I would especially like to thank Daniel Lustig and David Nellans for all of their support and guidance during the latter half of my Ph.D. study. I really enjoyed our weekly meetings and research discussions, which are invaluable research experience.

I cannot thank my girlfriend, Shisi Wang, enough for her great support during my Ph.D. She was always there no matter what happened. We shared all the highs and lows together. It is my fortune to be with her.

My colleagues have been important and helpful to the completion of my thesis. Bharath Pichai, Binh Pham, Jan Vesely, Guilherme Cox, Karthik Sriram, and Jae Woo Ju have all provided feedback, support, and entertainment through the years I spent in Rutgers.

Dedication

To Shisi, Mom, and Dad

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Thesis Statement

> High-performance virtual memory is achievable in tiered memory systems by accelerating page migration via hardware translation coherence and faster software page copying, and by mitigating address translation overheads via hardware/software co-design for TLB compression.

For decades, virtual memory has been vital to abstractions, mechanisms, and policies used to manage memory systems. With virtual memory, an application accesses data via its own virtual address space, abstracting away the complexity of the physical address space made up of a complex assortment of memory and storage devices. Two concepts are central to the success of modern virtual memory systems: paging and address translation.

Operating systems manage data at the granularity of pages of memory. Data accessed by applications are to locations within pages (e.g., usually contiguous 4KB chunks of virtual/physical address spaces on x86-64 systems). Several operations can trigger the act of paging or movement of pages of data. Data may be loaded into memory from disk, data may have to be written back from memory to disk, or data may have to be copied between memory devices. In tandem, OSes and hardware support virtual-to-physical memory address translation to realize virtual memory.

To implement paging and address translation efficiently, hardware and software support are required. On the software side, there is a need for good policies that measure attributes like page utility or hotness to determine which pages to place in faster memory, slower memory, or storage devices; efficient mechanisms to copy pages of data

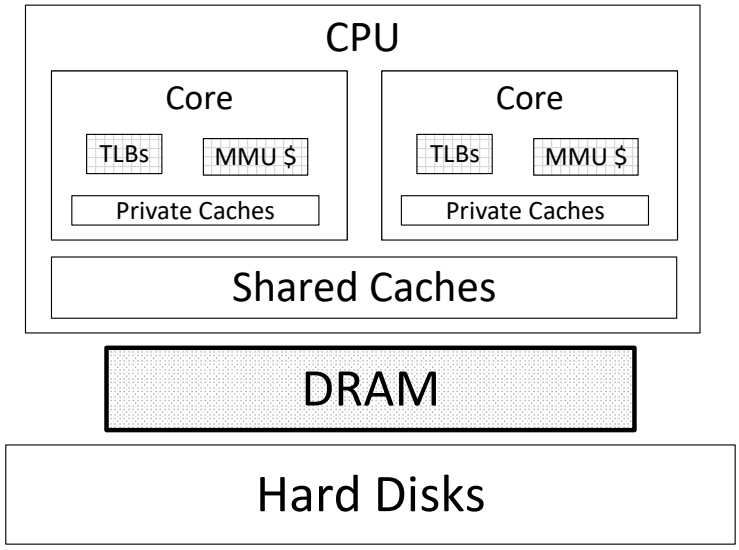
between memories and storage devices; and efficient means of allocating and remapping pages so as to improve the performance of hardware that is used to cache/maintain page table information. Meanwhile, on the hardware side, caches for virtual memory (i.e., TLBs, MMU caches, nested TLBs, etc.) must efficiently respond to page table management events. For example, as page tables increase in the number of entries they maintain, TLBs must be able to adapt their reach to ensure that the frequency of misses does not become too high. Additionally, per-core private TLBs, MMU caches, and nested TLBs must be kept coherent — just like private caches — to changes in the page table. Today's support for such events requires complex and often prohibitively expensive hardware/software cooperation.

This thesis is about the impact of emerging heterogeneous memory systems on these critical pieces of hardware and software support. Heterogeneous memory systems are those that maintain an array of distinct memory devices with varying latency, bandwidth, capacity, and persistence characteristics. Their objective is to provide processing elements access to a cost-effective memory system with the best attributes of many different device technologies. While heterogeneous memory systems hold great promise in the design of future high-performance systems (and are already seeing real-world adoption), they pose performance problems to the need for fast paging and address translation. We now describe in more detail these problems.

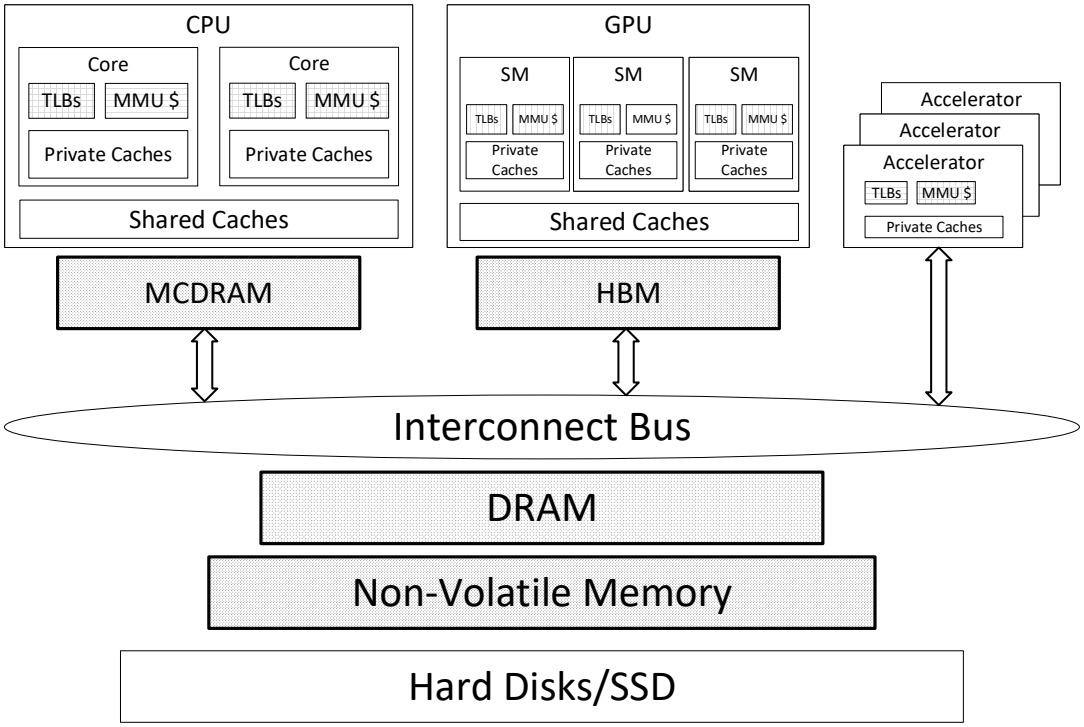
1.1 Memory Systems

Conventional systems employed homogeneous memory organizations, with a single layer of memory devices backed by storage technologies. Figure 1.1a exemplifies such a system, In this conventional system, a layer of memory is implemented via DRAM technology, which acts as a cache of a hard disk. Decades of research in architecture and operating system design has developed policies and mechanisms to best manage these layers of memory and storage to optimize for performance, power, availability, and reliability.

The recent development of heterogeneous memory systems, however, makes many

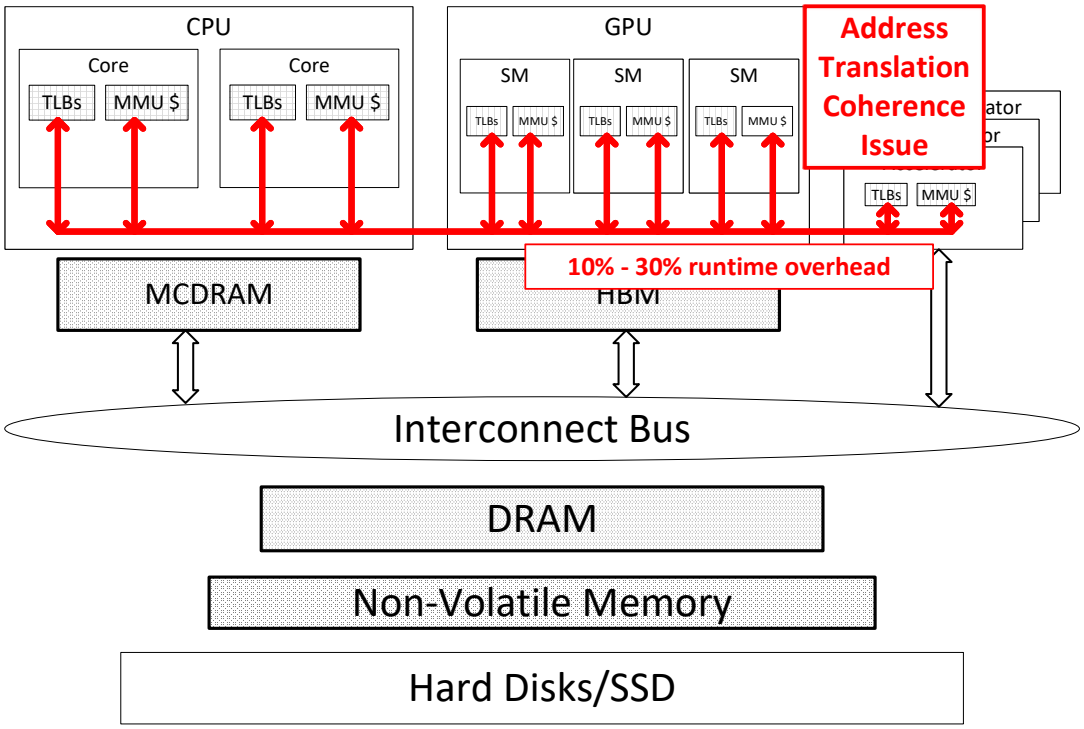


(a) Virtual memory for conventional homogeneous memory systems.

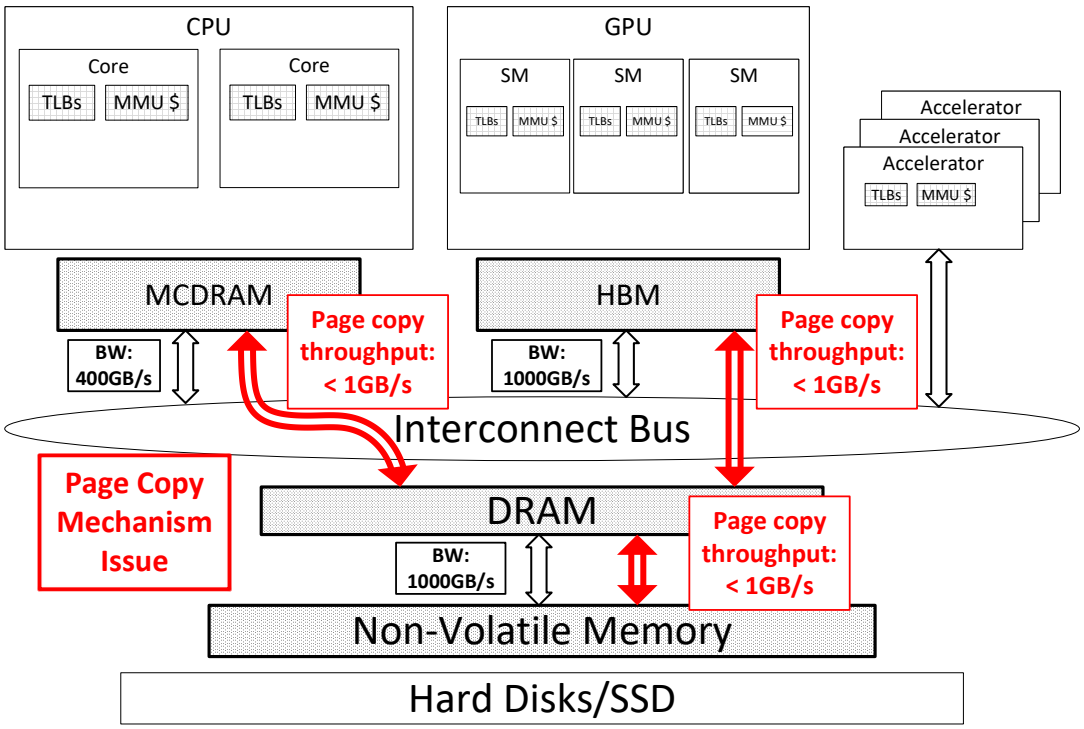


(b) Virtual memory for heterogeneous memory systems. I/O MMUs or Coherent Accelerator Processor Interface (CAPI) like devices, which add platform-level TLBs for accelerators in certain systems, are not shown for simplicity.

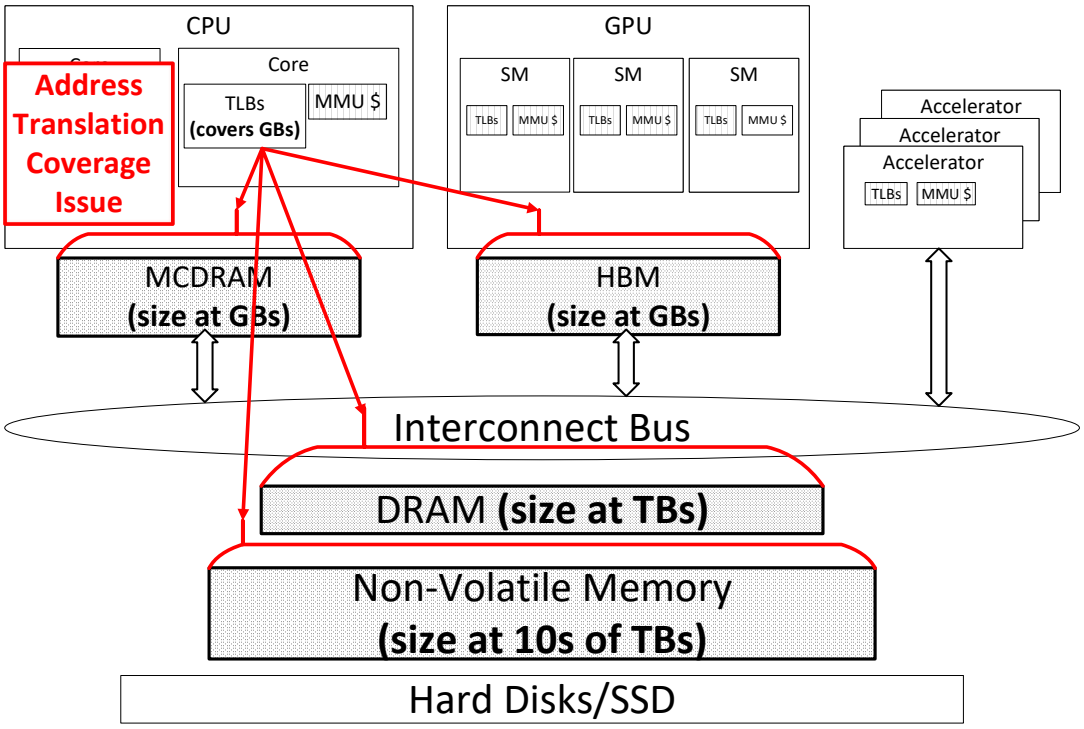
Figure 1.1: The evolution of address translation from conventional homogeneous memory systems to modern heterogeneous memory systems. Address translation structures are shown in grid-patterned boxes and different memory devices are shown in dot-patterned boxes. Multi-Channel DRAM (MCDRAM) and High-Bandwidth Memory (HBM) are high-bandwidth memory devices (2× to 10× bandwidth of DRAMs).



(a) Inefficient page migration due to high-overhead address translation coherence.



(b) Inefficient page migration due to low-throughput page copy mechanisms.



(c) High virtual memory overheads due to unmatched address translation coverage for high-capacity memories.

Figure 1.1: Three bottlenecks in virtual memory for heterogeneous memory management. a) High cost address translation coherence during page migration. b) Inefficient page copy mechanisms in operating systems. c) Excessive address translation coverage overheads from massive memory capacities.

of these conventional policies and mechanisms outdated or incomplete in some way. Heterogeneous memory systems integrate several memory technologies with differing latency, bandwidth, and capacity attributes into a single, cost-effective layer for the compute elements in the system. The exact composition and topology of such systems can vary. For example, it is possible that such systems are made up of high-bandwidth but capacity-constrained DRAM technologies (e.g., MCDRAM or HBM in Figure 1.1b), conventional DRAMs with lower bandwidth and higher capacity, and high-capacity but slower byte-addressable non-volatile memories (e.g., NVM also in Figure 1.1b). Furthermore, these memories can be organized in myriad ways, with some acting as logical caches of the others, and others acting as extensions of the overall physical address space. In other words, such systems offer "tiers" or levels of placement, which the OS and hardware have to account for in allocating/moving pages to the most desirable memory device from the perspective of the processing element that needs these pages. And since heterogeneous memory systems are arising mainly as a response to a sharp increase of heterogeneity in processing elements (e.g., emerging systems integrating not only CPUs but also GPUs, NICs, and hardware for video transcoding, computer vision, neural nets, etc.), the design space of how best to manage memory is a vast one to navigate. In this context, the critical mechanisms of page migration and address translation face significant challenges in the following ways.

Page migration: The two key challenges in page migration arise from sub-optimal page copy and translation coherence mechanisms. As the inter-memory-device bandwidth provided by hardware vendors continues to grow, the software support to move data between memories in commodity OSes becomes a bottleneck. Furthermore, as the number of computing elements and their associated virtual memory structures — i.e., TLBs, MMU caches, nested TLBs — proliferate, maintaining coherence amongst them becomes a challenging task.

Address translation: As memory technologies like non-volatile memory or 3D XPoint become common, hardware caches like TLBs are expected to increase their effective capacity to match the rapidly expanding overall size of addressable physical memory. Because TLBs and structures like MMU caches and nested TLBs are built close to

Issue Sources	Solution Types	Solution Techniques
Page copy mechanisms	Software	Parallelization, aggregation, and exchange
Address translation coherence	Hardware	Fuse with existing cache coherence protocols
Address translation coverage	Hardware and Software	Coalescing memory for emerging TLBs (Coalesced or Range TLB)

Table 1.1: Summary of the solutions to the virtual memory bottlenecks in heterogeneous memory management.

the pipelines within processing elements, it is difficult to increase their size efficiently. Therefore, such high-capacity tiered memory systems risk increasing TLB miss frequency, hence harming performance.

1.2 Research Goals

This thesis focuses on upgrading OS support for page migration, hardware support for translation coherence, and OS support to encourage the use of compression in TLBs. My work shows that efficiently architecting solutions to these three approaches has the potential to improve the performance of tiered memory systems significantly. A key theme of this thesis is to use modest hardware, software, or hardware/software co-design to achieve these goals. These techniques and approaches are summarized in Table 1.1.

To evaluate our approach in system environments representative of real-world deployments, most of our work is performed on real hardware running stock Linux. When necessary, we create performance models of hardware structures and rely on simulations to project performance improvements from our proposed novel hardware. Additionally, when possible, we have merged some of our research implementations into the upstream Linux kernel. The remainder of this chapter outlines the approach this work takes.

1.3 Fast Page Migration

Page migration has been an important topic of research for several decades, even for traditional homogeneous memory systems and later, non-uniform memory access or NUMA systems. However, while we focus on the mechanisms that enable page migration, the vast majority of prior work focused on the policies used to decide when to affect page migration. There are several reasons for this change in focus. On the earliest non-NUMA homogeneous memory systems, the transfer rate difference between memory and disk were so high that any additional page movement overheads were negligible in comparison; consequently, intelligently deciding when to migrate (and to minimize migration frequency) was the more critical question [41, 102, 148]. Going further, even the transition to NUMA systems placed the approach on policies that opted to migrate processes close to their data, rather than the other way around [34,50,86]. And finally, a general perception existed that for these systems – and the relative infrequency of page migrations – existing mechanisms were sufficiently performant [4,22,24,26,111,112,150].

This reality is markedly different for heterogeneous tiered memories. First, to leverage the benefits of various memory devices, pages have to be dynamically moved amongst the devices. Second, the idea of using process migration as an alternative to page migration is now invalid; i.e., all tiers of memory are accessible to all processing elements, many without discernible differences in access time. In such situations, intelligently migrating hot pages to memory devices, for example, that might have higher bandwidth are key to good performance. In other words, rather than being used rarely when other techniques fail, page migrations now become essential to overall system performance.

To implement a fast page migration, two major bottlenecks need to be addressed: high-overhead address translation coherence and low-throughput data copy operations. Address translation coherence, which ensures that all cached translation information in TLBs and other hardware translation structures are synchronized with the operating system page tables, must be performed on each migration to ensure correctness. Meanwhile, the data copy portion of page migration has long been unoptimized because of

the relative infrequency of its use. This thesis attacks these problems with modest hardware and software changes.

1.3.1 Low Overhead Hardware Translation Coherence

When a page is migrated, its virtual-to-physical page mapping in the page table must be changed by the OS. Consequently, the OS must also initiate translation coherence operations (e.g., TLB shootdowns in x86 systems) to invalidate stale virtual-to-physical page mappings that may be resident in any compute element's TLB, MMU cache, or nested TLB structures. Past work has shown that these operations can be expensive, taking up to 10-30% of system runtime [116, 134, 154]. These overheads are becoming even worse for modern systems that have increasing core counts (with more TLBs, MMU caches, and nested TLBs to keep coherent), a greater diversity of accelerators (for many of which even a single TLB invalidation can be catastrophic for performance), and virtualization (where multiple levels of page tables are maintained, complicating translation coherence substantially). Measurements on real hardware show that in virtualized systems, translation coherence activity can consume as much as 40% of system runtime.

Chapter 2 presents our approach to mitigating this problem via Hardware Translation Invalidation and Coherence (HATRIC). HATRIC is a hardware translation coherence scheme for both native systems and virtualized systems [163]. The key idea is to extend translation structures with coherence tags (or co-tags) and use them to piggyback translation coherence atop cache coherence. Not only does HATRIC work on native systems, but it also eliminates the translation coherence overhead on virtualized systems (a topic that has mostly been ignored by prior work [116, 134, 154]). In more detail, HATRIC precisely invalidates stale translation information in all translation structures with the help of co-tags; it accurately identify the victim CPUs by augmenting existing cache coherence protocols; it performs all invalidation operations in hardware atop the cache coherence protocol, which avoids expensive software operations, especially the more costly ones in virtualized systems. As a result, HATRIC

minimizes address translation coherence overheads during the process of all page migrations.

1.3.2 High Throughput Operating System Page Migration

In addition to address translation coherence, page migration overheads arise from inefficient copy operations. This is because page copy operations are still implemented using single-threaded, serialized, and unoptimized software approaches. It is particularly exigent to upgrade these antiquated page copy mechanisms because the hardware bandwidth between memory devices in emerging tiered memory systems is considerably higher than what software currently exploits. Taking Linux as an example, a page is migrated in five steps: ① allocating a new page, ② unmapping the existing virtual to physical address translation and performing address translation coherence, ③ copying data from the old physical page into the new physical page, ④ mapping the virtual address to the new physical page, and finally ⑤ freeing the old physical page. The actual page copy process occurs in step ③, which is a single-threaded and unoptimized process, whereas steps ①–② and ④–⑤ are runtime correctness guarantees inside the operating system kernel and have been carefully tuned.

Chapter 3 presents four composable page migration optimizations. First, transparent huge page migration support moves a single huge page between memory devices, instead of individually moving multiple base pages one at a time. This reduces kernel overheads for each page migration. Second, parallel page migration enables parallelization of page copy to enhance data copy throughput. Third, concurrent page migration aggregates page copy operations to increase data copy throughput further. Finally, page exchange eliminates extra kernel operations by merging two symmetric page migrations into one. All these optimizations leverage existing hardware and can be easily adopted in the existing operating systems. Specifically, my work merges support for transparent huge page migration support into the Linux upstream kernel v4.14 [110]. In total, the combination of these four optimizations improves existing Linux page migration throughput by 15×. This throughput improvement ultimately yields 40% performance increases on our disaggregated memory evaluation testbed.

1.4 Scalable Address Translation with Fast Page Migration

The emergence of memory technologies that permit stacking of high-bandwidth DRAM on CPUs or byte-addressable non-volatile memories has the potential to increase overall memory capacity on modern systems dramatically. While offering the prospect of significant performance improvements for memory-intensive workloads, it also highlights a growing problem with address translation — the inability of current TLB, MMU cache, and nested TLB hardware capacities to keep up with the growth of memory capacity.

My work notes that a major source of poor address translation performance arises from the fact that conventional virtual memory uses fixed-sized pages (e.g., x86_64 uses 4KB, 2MB, and 1GB) as basic address translation units. Unfortunately, this means that there exist gaps between page sizes and that there is a lack of support for page sizes that exceed 1GB. Several recent studies have observed this problem and in response, have proposed TLB designs that leverage relatively simple incarnations of compression approaches [13, 76, 121, 125]. In particular, these approaches rely on the notion of "translation contiguity", where a set of contiguous virtual pages map to a corresponding set of contiguous physical pages. Such contiguity enables a single TLB entry to map all the virtual-to-physical mappings. In other words, high address translation coverage is achieved by compressing multiple page table entries into one TLB entry, whether they are for 4KB, 2MB, or 1GB pages [121, 125].

To date, however, such compression schemes have been shown to be applicable in a limited set of real-world environments. In particular, it is challenging for a system to produce arbitrary amounts of contiguity when the memory is fragmented in long-running systems. To fully exploit the benefits of using contiguity-aware TLB designs, there is a need for robust OS support to generate contiguity even in highly loaded fragmented systems. Chapter 4 presents such an operating system service, Translation Ranger, to actively coalesce memory to produce unbounded amounts of translation contiguity in the presence or absence of memory fragmentation. Unlike existing techniques that try to generate translation contiguity at page allocation time [13, 76, 121, 125], Translation Ranger coalesces memory post page allocation. This method affords greater

flexibility and opportunity in producing translation contiguity.

By leveraging our previous work on faster page migration, Translation Ranger can be implemented efficiently in modern operating systems. We implement Translation Ranger in Linux v4.16 to assess the feasibility of the approach. We quantify both the amount of translation contiguity generated by Translation Ranger and the overheads incurred by the execution of it. Translation Ranger creates significant translation contiguity ($> 90\%$ of 120GB application footprint covered by only 128 contiguous regions, compared to $< 1\%$ without it) and costs low runtime overheads ($< 2\%$ of overall application runtime with 120GB application memory footprint). Both together result in a net win of coalescing memory, when combining with any available TLB designs that take advantage of translation contiguity by minimizing address translation overheads.

1.5 Contributions

This thesis contributes the following:

- To reduce the overheads of translation coherence to accelerate page migration, this thesis proposes Hardware Translation Invalidation and Coherence (HATRIC) to replace existing software approaches for TLB coherence. HATRIC adds coherence-tags in translation structures (e.g., TLBs) and uses them to leverage messages picked up by existing cache coherence protocols. Extending cache coherence protocols in this manner to support native and virtualized systems improves performance and energy-efficiency. Importantly, this thesis shows that translation coherence overheads become a much bigger problem in the context of heterogeneous memory management and virtualization.

- To further improve page migration from software aspect, this thesis proposes four page migration optimizations and implements them in Linux. These optimizations achieve 15× page migration throughput boost over the existing Linux page migration. Further integrating all four optimizations into Linux for a disaggregated memory system improves average application performance by 40%.

Importantly, this work corrects a misconception that heterogeneous memory management requires only good policies and can ignore the mechanism for page migration. Furthermore, one of the page migration optimization, transparent huge page migration, has been merged into Linux, a modern and commercial operating system.

- To counteract the rising cost of address translation, this thesis proposes a novel operating system service that actively coalesces memory to generate unbounded amounts of address translation contiguity to minimize address translation overheads and make virtual memory scalable with ever-increasing memory sizes. For 120GB application footprint, only 128 contiguous regions, which come at the cost of $< 2\%$ of runtime overhead, can cover $> 90\%$ of total footprint. This service permits coalescing memory at post-memory allocation time regardless of the extent of system memory fragmentation. In other words, robust use of translation contiguity becomes possible.

1.6 Dissertation Organization

The rest of this thesis is organized as follows. Chapter 2 introduces hardware translation coherence and evaluates the proposed approach in a die-stacked memory system with virtualization. Chapter 3 optimizes page migration along and evaluates its utility with a heterogeneous memory management policy for a disaggregated memory system. Chapter 4 implements an OS service for coalescing memory and present real hardware results for both address translation contiguity measurements and overall runtime overheads of this service. Finally, Chapter 5 concludes this thesis.

Chapter 2

Hardware Translation Coherence for Virtualized Systems

2.1 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 [64]. AMD and Hynix are releasing High-Bandwidth Memory or HBM [21, 79]. Similarly, Micron's Hybrid Memory Cube [122, 140] and byte-addressable persistent memories [39, 131, 160, 161] 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 [3, 9, 79, 103, 116, 123, 131, 153].

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 [8, 81, 109, 147], to migrate pages among NUMA sockets [50, 86], and to deduplicate memory by enabling copy-on-write optimizations [126, 127, 138]. 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 [32, 52, 78].

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 [19, 96, 124, 125], MMU caches [16], 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 [116, 134, 154]. 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 [15]. 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 [145] and has developed efficient hardware cache coherence protocols [100]. 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.

① **Precise invalidation:** Processors use several hardware translation structures – TLBs, MMU caches [11, 16], and nested TLBs (nTLBs) [15] – 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) [154]. 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 [5, 15, 18, 27, 36, 49, 124, 125, 127].

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 [2, 15].

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 [134], 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 [116, 154], 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 [116, 134, 154] 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 [134] 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 [116,154]) 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 [5,49] and native execution.

2.2 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.

2.2.1 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 [5]. However, the overheads of page table synchronization can be high [49]. Consequently, most systems now use two-dimensional page tables instead. Figure 2.1

illustrates two-dimensional page table walks (see past work for more details [5, 11, 12, 15, 48, 127]). 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 [12, 15, 48, 127]. We refer to these as levels 4 (the root level) to 1 (the leaf level) similar to recent work [11, 15, 16]. 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 [11, 15, 16]. There are two flavors of MMU cache. The first is a *page walk cache* and is implemented in AMD chips [15, 16]. Figure 2.1 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 [11, 16]. 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 [11, 16].

③ **Private per-CPU nTLBs** short-circuit nested page table lookups by caching GPP to SPP translations [15]. Figure 2.1 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).

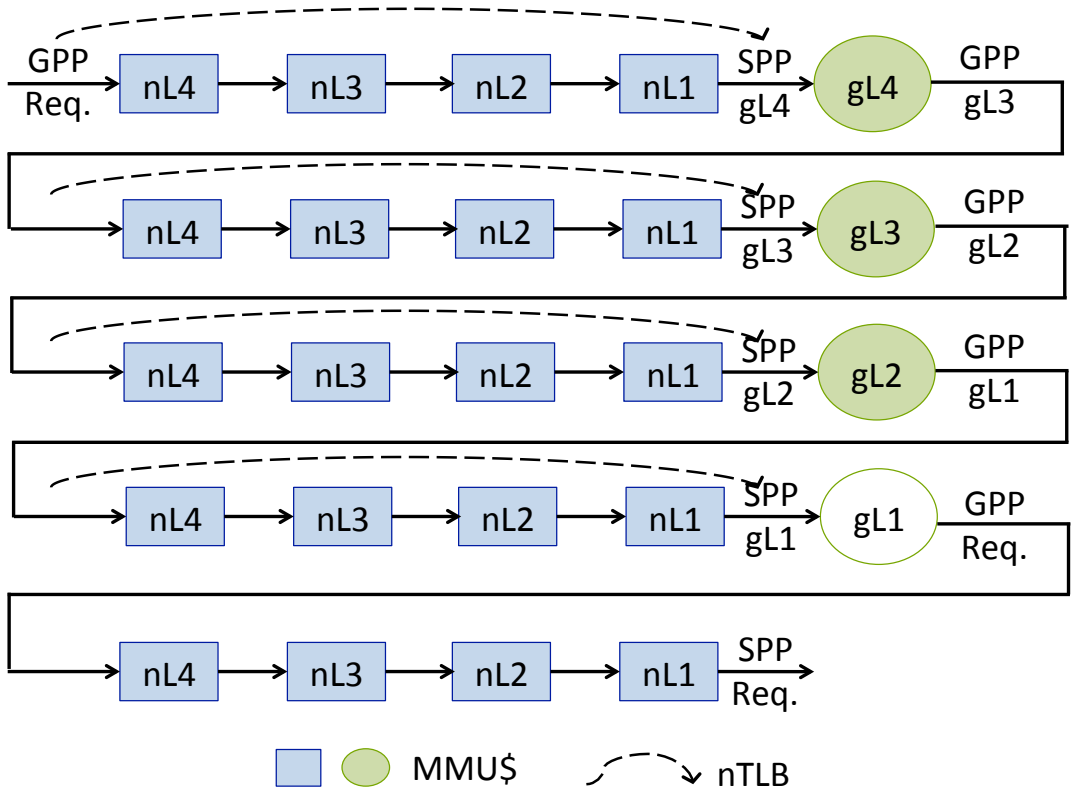


Figure 2.1: Two-dimensional page table walks for virtualized systems. Nested page tables are represented by boxes and guest page tables are represented by circles. Each page table's levels from 4 to 1 are shown. We show items cached by MMU caches and nTLBs. TLBs (not shown) cache translations from the requested guest virtual page (GVP) to the requested system physical page (SPP).

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.

2.2.2 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 [117]. 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 [57,127] and page migration between NUMA memories on multi-socket systems [10, 132, 133] 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 [116]).

2.3 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 [74,116]. 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 [44, 92, 116, 155].

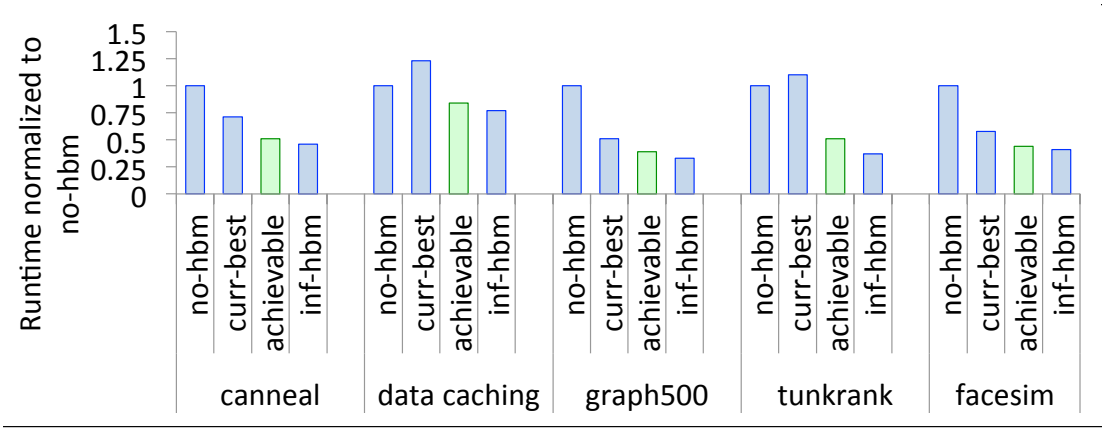


Figure 2.2: Performance of no-hbm (no die-stacked DRAM), inf-hbm (data always in die-stacked DRAM), curr-best (best die-stacked DRAM paging policy with current software translation coherence overheads), and achievable (best paging policy, assuming no translation coherence overheads). Data is normalized to no-hbm runtime.

2.3.1 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 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 [116]. 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 2.5. Like prior work [116], 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 2.2 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 2.6), 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 2.2 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 2.2 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 [116]. 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.

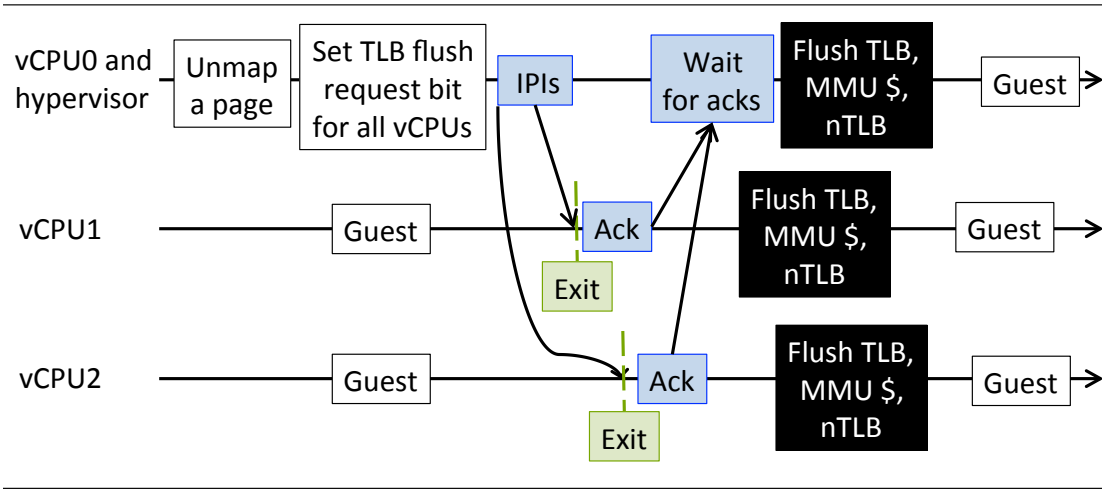


Figure 2.3: Sequence of operations associated with a page unmap. Initiator to target IPIs are shown in blue ①, VM exits are shown in green ②, and translation structure flushes are shown in black ③.

2.3.2 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 2.3 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 2.3), 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 2.3 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 [116, 154], 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 2.1. 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.

2.3.3 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.

2.4 Hardware Design

We now detail HATRIC's design. HATRIC achieves all three goals from Section 2.1. 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.

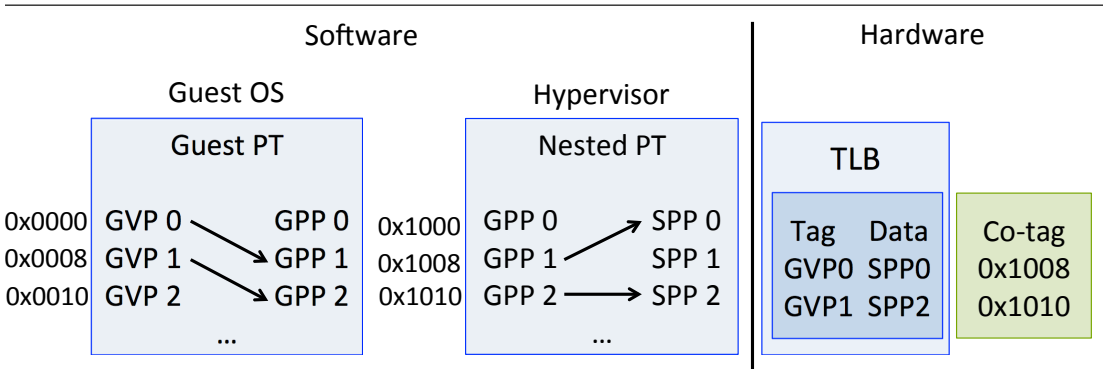


Figure 2.4: We add co-tags to store the system physical addresses where nested page table entries are stored. In our final implementation, we only store a subset of the system physical address bits.

2.4.1 Co-Tags

What are co-tags? Consider the page tables of Figure 2.4 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 2.1). 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 [134].

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 [45, 71, 73, 77, 144], 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 2.6 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.

2.4.2 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 [164].

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.

2.4.3 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.

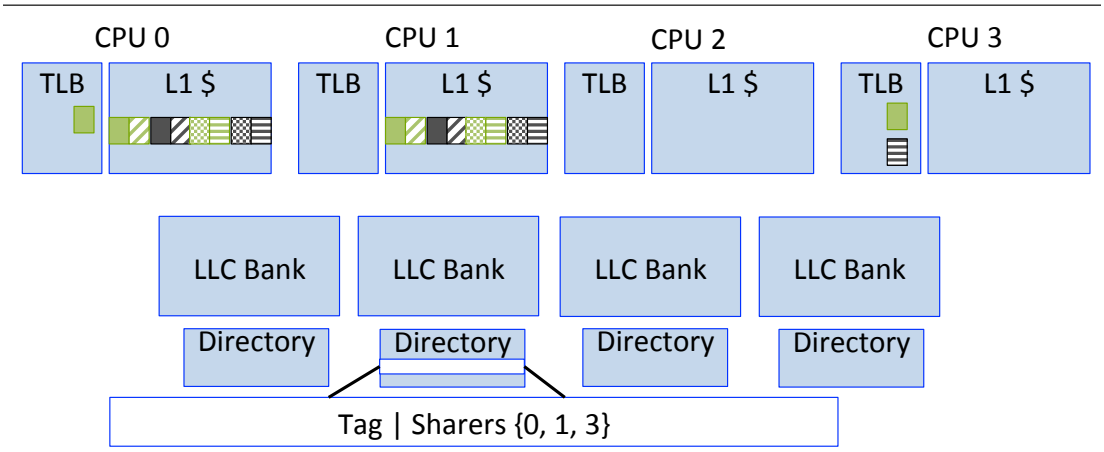


Figure 2.5: Coherence directories identify translation structures caching page table entries, aside from private L1 cache contents.

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 2.5 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 2.5, 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 2.6).

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 2.5, 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 2.4.1, 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 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 2.6), we use two additional directory coherence directory entries, which comfortably maintain all 11 set identification bits per page table entry.

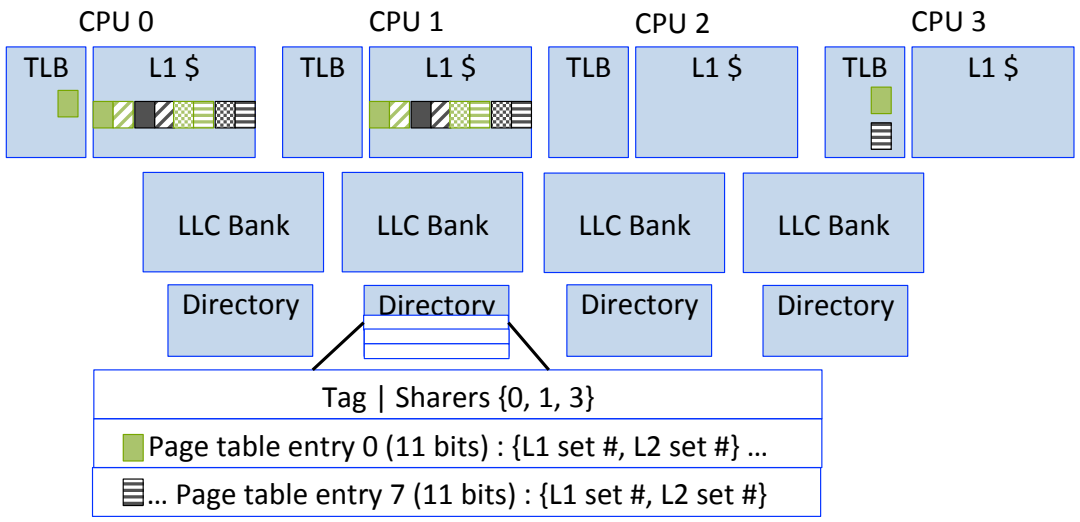


Figure 2.6: We use additional coherence directory entries adjacent to the directory entry tracking sharers, to track TLB set numbers.

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 2.7 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

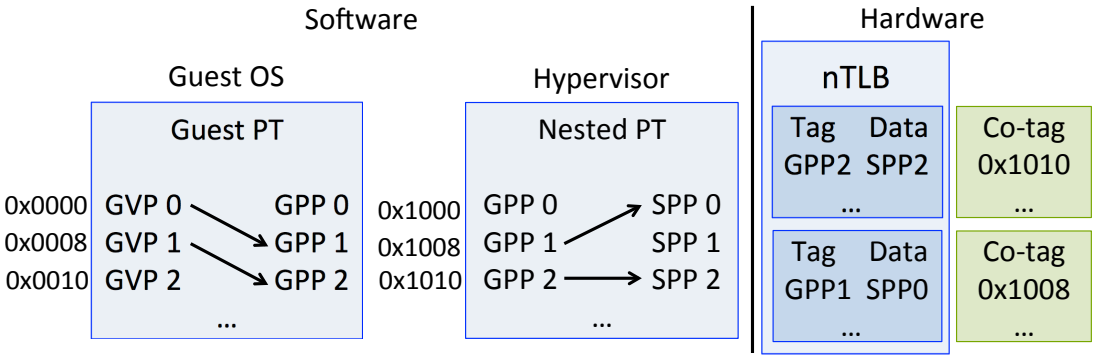


Figure 2.7: MMU cache and nTLB set numbers can be inferred directly from the physical address of the page table entry being modified, so there is no need to store them in the coherence directory entries.

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 2.7) today, there is no need to embed MMU cache or nTLB set numbers in the coherence directory entries.

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

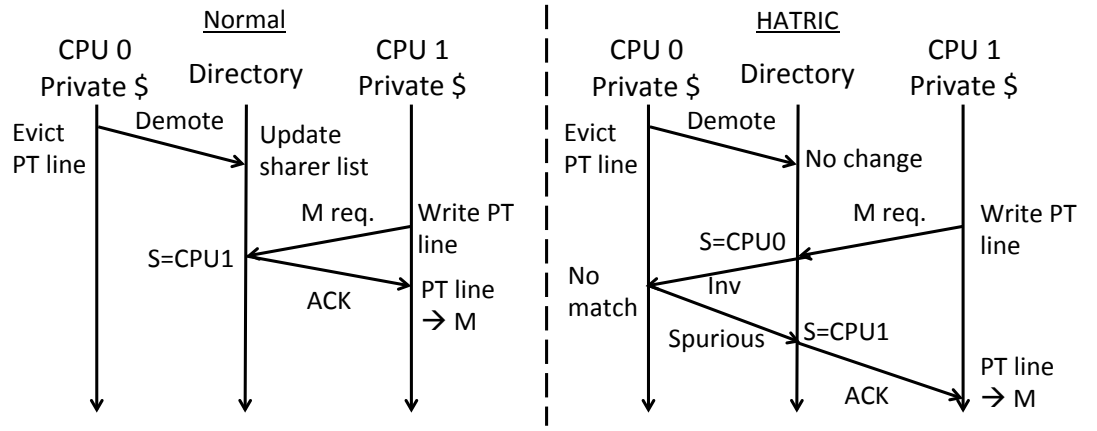


Figure 2.8: Coherence activity from the eviction of a cache line holding page table entries from CPU0's private cache. HATRIC updates sharer list information lazily in response to cache line evictions.

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 [164]. 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 2.8 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 [145]. To showcase this in detail, suppose CPU 0 evicts a cache line containing page table entries.

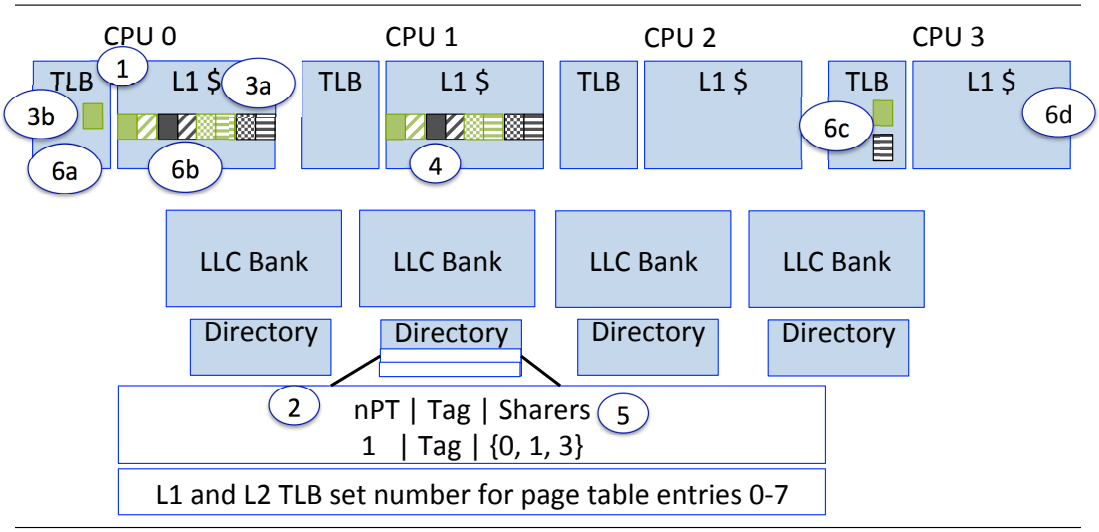


Figure 2.9: Coherence directories identify translation structures caching page table entries, aside from private L1 cache contents.

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 [164]. 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.

2.4.4 Putting It All Together

Figure 2.9 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 2.1 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 2.9) 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.

2.4.5 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 [127]. 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 [97], 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 [134]. HATRIC uses energy-frugal co-tags instead of UNITD's large reverse-lookup CAMs, achieving greater energy efficiency. We showcase this in Section 2.6 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 [154] also targets translation coherence for non-virtualized systems. Similarly, recent work investigates translation coherence overheads in the context of die-stacked DRAM [116]. 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 [117], while HATRIC tackles nested page table coherence.

2.5 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.

2.5.1 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 [116]. Each CPU maintains 32KB L1 caches, 256KB L2 caches, 64-entry L1 TLBs, 512-entry L2 TLBs, 32-entry nTLBs [15], and 48-entry paging structure MMU caches [16]. Further, we assume a 20MB LLC. We model the energy usage of this system using the CACTI framework [107]. 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 [95] 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 [116] 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 [125, 127], 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.

2.5.2 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 [41]. 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 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.

2.5.3 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 [20], data caching and tunkrank from Cloudsuite [46], and graph500 as part of this group. We also create 80 multiprogrammed combinations of workloads from all the SPEC applications [60] 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 [20] and SPEC applications [60] for these studies.

2.6 Evaluation

Performance as a function of vCPU counts: Figure 2.10 shows HATRIC's runtime, normalized as a fraction of application runtime in the absence of any die-stacked memory (no-hbm from Figure 2.2). 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

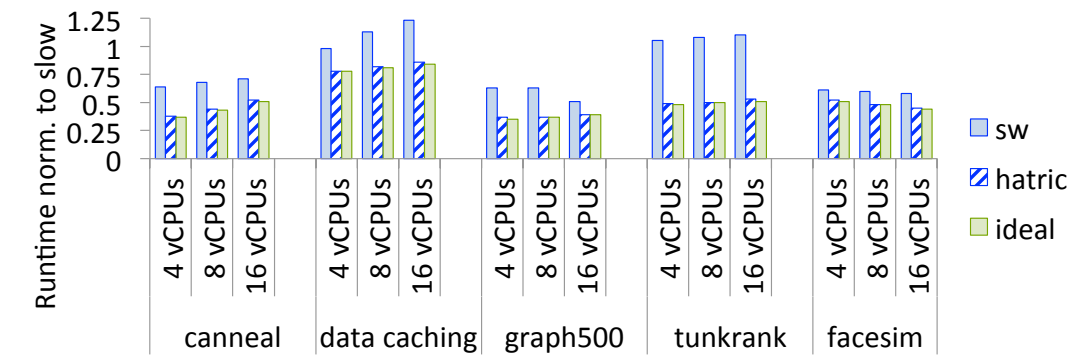


Figure 2.10: For varying vCPUs, runtime of the best KVM paging policy without HATRIC (sw), with HATRIC (hatric), and with zero-overhead translation coherence (ideal). All results are normalized to the case without die-stacked DRAM.

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 2.10 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 2.11 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 2.11 shows HATRIC improves runtime substantially for any paging policy. Performance is best when all techniques are combined but HATRIC achieves 10-30%

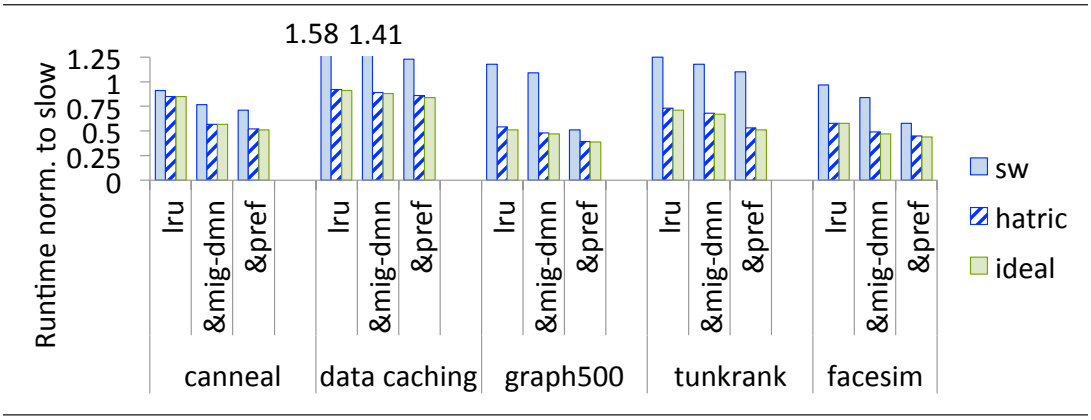


Figure 2.11: HATRIC's performance benefits for KVM paging policies, with LRU, migration daemons (mig-dmn), and prefetching (pref.). Results are normalized to the case without die-stacked DRAM.

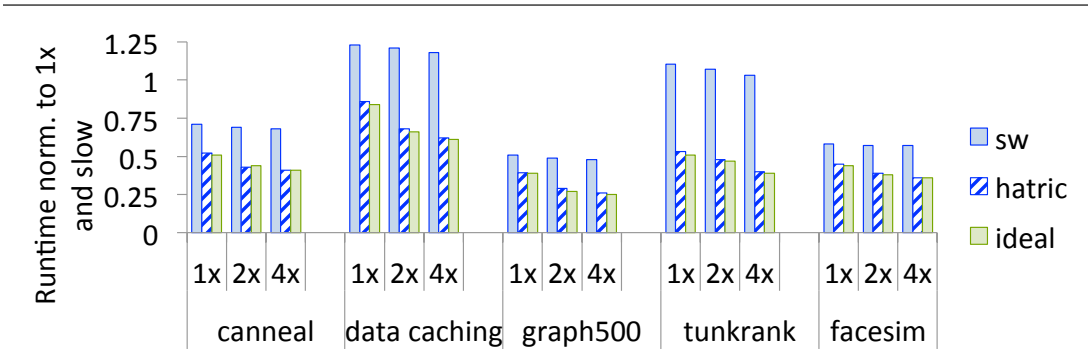


Figure 2.12: HATRIC's performance benefits as a function of translation structure size. 1× indicates default sizes, 2× doubles sizes, and so on. All results are normalized to the case without die-stacked DRAM.

performance improvements even for just lru. Furthermore, Figure 2.11 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

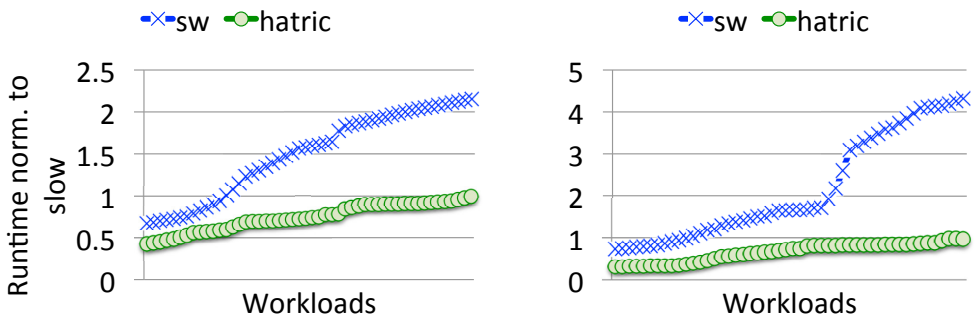


Figure 2.13: (Left) Weighted runtime for all 80 multiprogrammed workloads on VMs without (sw) and with HATRIC (hatric); (Right) the same for the slowest application in mix.

as translation structures become bigger (and flushes needlessly evict more entries). Figure 2.12 quantifies the relationship. We vary TLB, nTLB, and MMU cache sizes from the default (see Section 2.5) to double (2×) and quadruple (4×) the number of entries.

Figure 2.12 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 2.12 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 [146]. 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 2.13 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 2.12 shows the runtime of the slowest application in the workload mix; again, lower runtimes indicate a speedup in the slowest application.

Figure 2.13 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.

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).

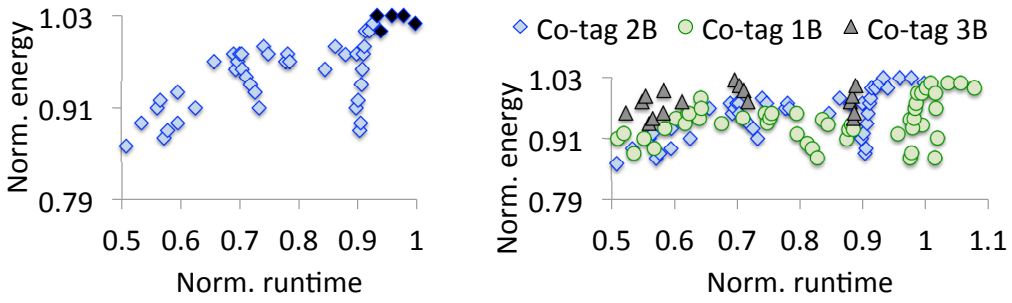


Figure 2.14: (Left) Performance-energy plots for default HATRIC configuration compared to a baseline with the best paging policy; and (Right) impact of co-tag size on performance-energy tradeoffs.

The graph on the left of Figure 2.14 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 2.14 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 2.14 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,

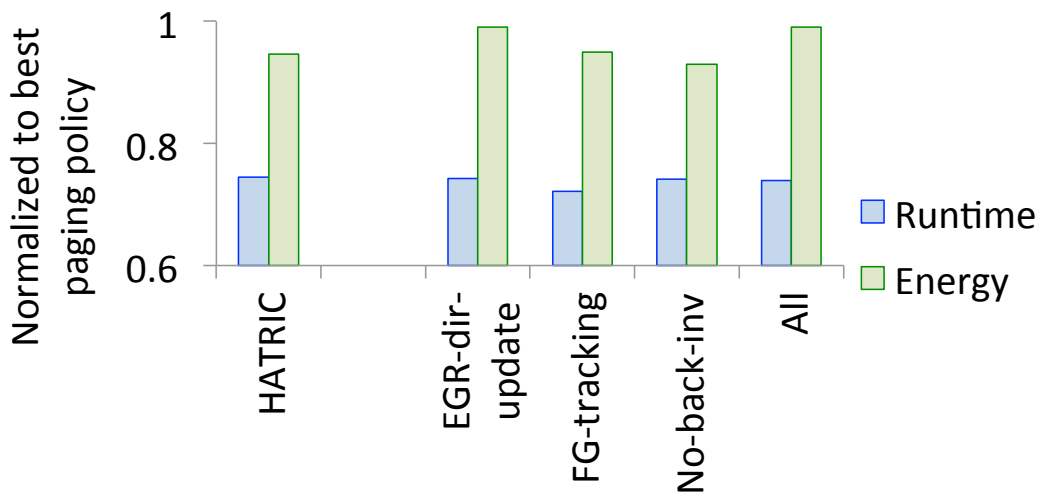


Figure 2.15: Baseline HATRIC versus approaches with eager update of directory on cache and translation structure evictions (EGR-dir-update), fine-grained tracking of translations (FG-tracking), and an infinite directory with no back-invalidations (No-back-inv). All combines these approach. We show average runtime and energy, normalized to the metrics for the best paging policy without HATRIC.

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.

Coherence directory design decisions: Section 2.4 detailed the nuances modifying traditional coherence directories to support translation coherence. Figure 2.15 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 2.15 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 2.15 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 2.15 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 [134]. 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 page tables entries in the originally proposed reverse-lookup CAM [134]. Second, we extend UNITD to work seamlessly with coherence directories. We call this upgraded design UNITD++.

Figure 2.16 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

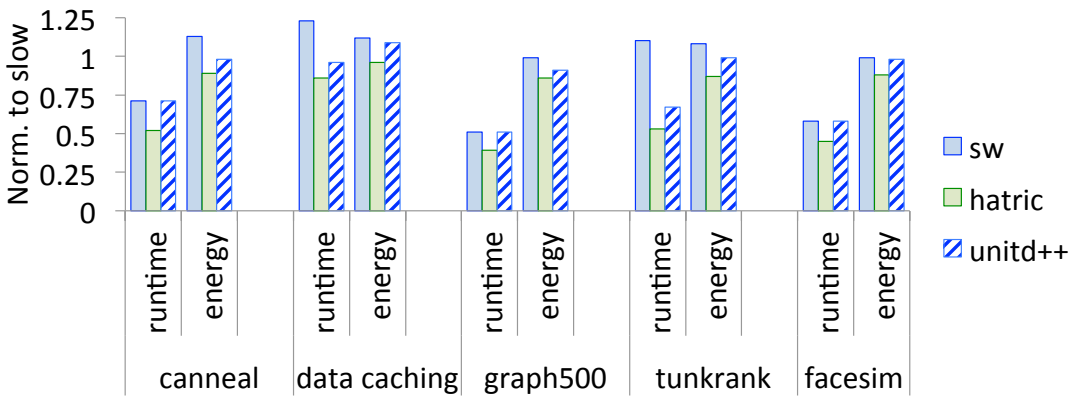


Figure 2.16: Comparison of HATRIC's performance and energy versus UNITD++. All results are normalized to results for a system without die-stacked memory and compared to sw.

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.

2.7 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.

Chapter 3

Nimble Page Management for Tiered Memory Systems

3.1 Introduction

Modern computing systems are embracing heterogeneity in their processing and memory systems. Processors are specializing to improve performance and/or energy efficiency, with CPUs, GPUs, and accelerators pushing the boundaries of instruction and data level parallelism. Memory systems are combining the best properties of emerging technologies that may be optimized for latency, bandwidth, capacity, or cost. For example, Intel's Knight's Landing uses a form of high bandwidth memory called multi-channel DRAM (MCDRAM) alongside DDR4 memory to achieve both high bandwidth and high capacity [65, 67]. Non-volatile 3D XPoint memory has been commercialized for next-generation database systems, and disaggregated memory may be a promising solution to capacity scaling for blade servers [62, 87]. Both CPUs and GPUs are embracing heterogeneous memory with IBM and NVIDIA having recently delivered supercomputers containing high-bandwidth GPU memories and high-capacity CPU memories [68, 84, 114, 115].

Figure 3.1 illustrates an abstract example of the memory systems architects and OS designers will likely have to consider in the future. These systems consist of a compute node (CPU, GPU, or both) connected to multiple types of memory with varying latency, bandwidth, and/or capacity properties. Of course, the particular configuration will vary by system.

The critical operating system support needed to enable the vision of efficiently moving data as programs navigate different phases of execution, each with potentially

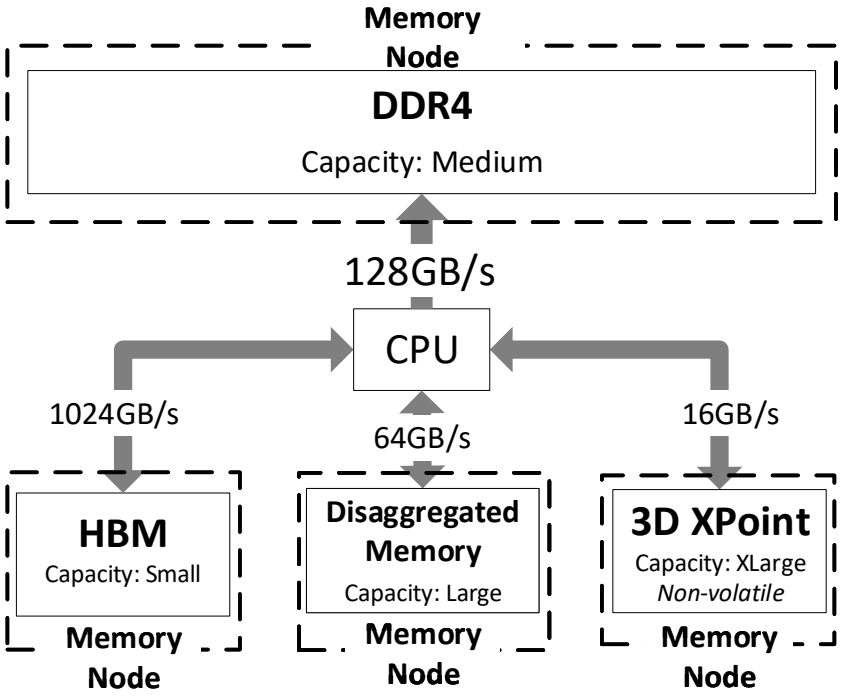


Figure 3.1: A hypothetical future multi-memory system with 4 technology nodes, all exposed as non-uniform memory nodes to the operating system.

distinct working sets, is *efficient page management and migration*. Regardless of configuration, to optimize for performance, ideally the hottest pages will be placed in the fastest memory node (in terms of latency or bandwidth) until that node is filled to capacity, the next-hottest pages will be filled into the second-fastest node up to its capacity, and so on. Then as programs execute, these pages must be constantly re-organized based on their hotness to retain maximum workload performance.

Unfortunately, page migration in today's systems is surprisingly inefficient. We performed an experiment in which we moved pages between two memory nodes where each local node has more memory bandwidth available than the inter-socket interconnect. After allocating memory from distinct memory nodes, we measured both the cost breakdown and the throughput of several types of cross-socket page migration. Figure 3.2 shows the cost breakdown and throughput achieved when migrating different page sizes on Linux today. For single base page migration, the majority of time is spent

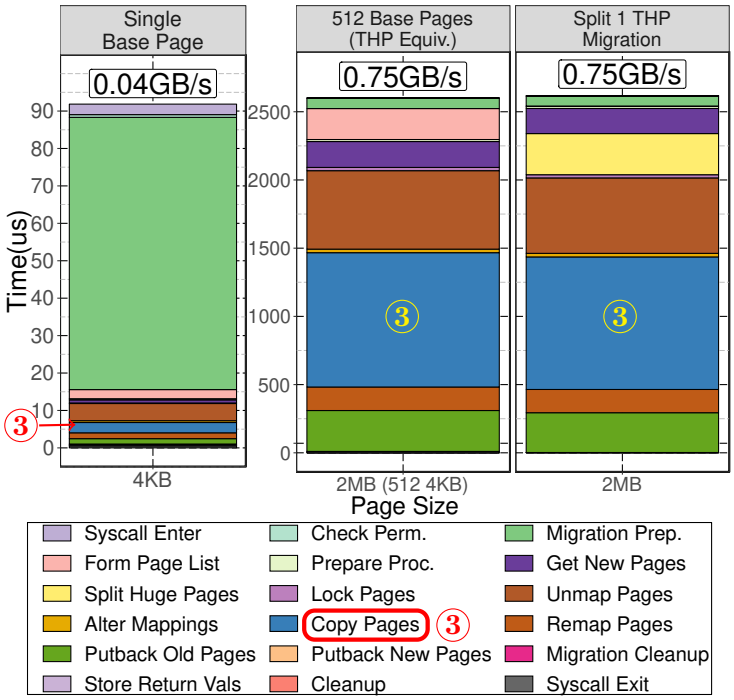


Figure 3.2: Page migration cost breakdown for migrating a single 4KB page, 512 consecutive base pages, and both splitting and migrating a 2MB THP. (Figure best viewed in color.)

within kernel memory management and synchronization routines. Only a small fraction of time is consumed by the actual page copy. As a result, the effective migration throughput is just 40MB/s even though the hardware that has 19.2GB/s cross-socket memory bandwidth (see Table 3.1 for our experimental platform configuration). We also scaled the number of pages being migrated to 512, matching the huge page size (2MB), to amortize the software overhead across multiple migrations. In this case, throughput does scale up to 750MB/s, but this is still just 5% of peak hardware bandwidth. To investigate the potential improvement of page migration, we profile the data copy throughput of our system (described in Section 3.4.1) with different thread counts and data sizes including 2MB. Figure 3.3 shows that existing page migration throughput is 10× slower than what is achievable with a 2MB data size and the gap is bigger with larger transferred data sizes.

To eliminate the page migration bottleneck, we propose a set of four optimizations:

transparent huge page migration, *parallelized data copy*, *concurrent multi-page migration*, and *symmetric exchange of pages*. On top of these mechanisms we build a holistic multi-level memory solution that directly moves data between heterogeneous memories using the existing OS active / inactive page lists, eliminating another major source of software overhead in current systems. The novel contributions of our work are as follows:

1. We show that breaking transparent huge pages (THPs), which are currently non-movable, into movable base pages reduces the effectiveness of THPs. We also demonstrate that existing base page migration only achieves throughput an order of magnitude lower than hardware line rate. We remedy this by implementing native huge page (THP) migration which improves migration throughput by 2.8× over the Linux baseline, while also having the side effect of improving TLB coverage.

2. Our additional page migration optimizations improve throughput by 5.2× over our native THP migration implementation alone. Together, this set of improvements increases page migration throughput 15× over the state of the art today. By reusing existing OS interfaces, these optimizations are automatically inherited by any memory management policy using the standard Linux memory management APIs. Our claim of broad applicability is supported by the fact that some of these improvements have been adopted into upstream Linux [110].

3. We explore a simple, yet general, end-to-end heterogeneous memory profiling and placement policy. Unlike existing implementations and other recent proposals, our proposed system does not swap data out to disk, nor does it make portions of memory unavailable to profile accesses to them via page faults. Instead, it simply repurposes the existing OS active/inactive page lists. Therefore, our approach imposes no profiling overhead on systems which may not need its functionality.

4. We show that in a disaggregated memory system, an emerging class of important multi-tier memory system [55, 87, 88], our optimized OS support for multi-level

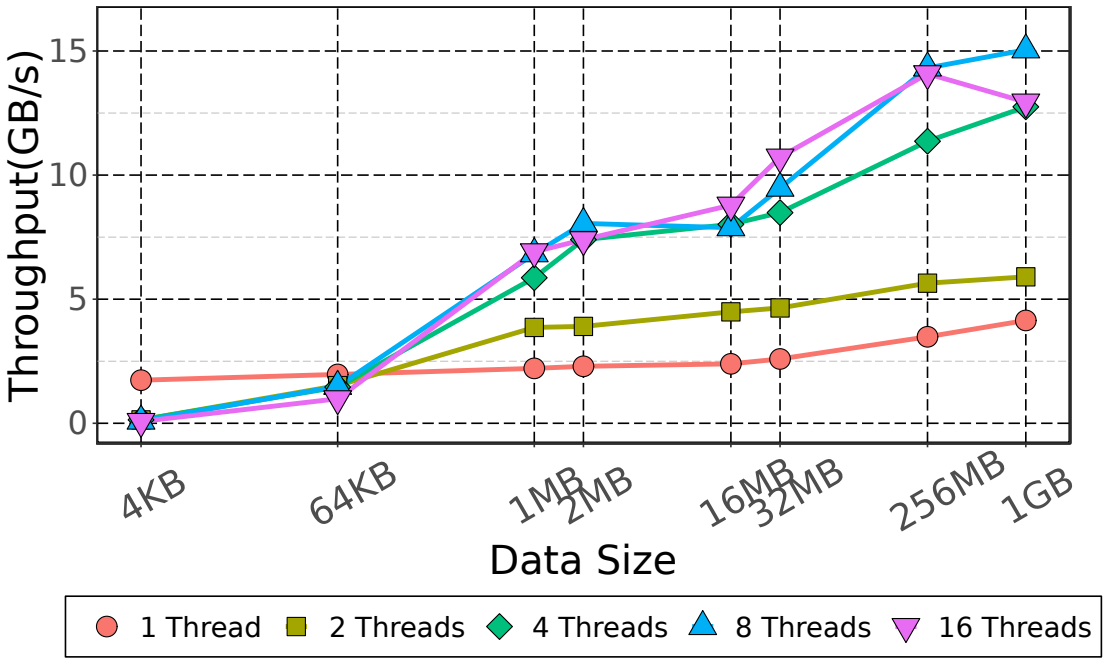


Figure 3.3: Impact of thread count and transfer size on raw data copy throughput (**higher is better**).

memories will improve application performance over 40%, on average, compared to current OS support for multi-level memories.

3.2 Background

Modern heterogeneous memory systems typically consist of low-capacity, high-bandwidth memory as well as high-capacity, low-bandwidth memory. Latency to large capacity memories is also expected to be higher due to longer, potentially multi-hop physical connections or differences in the underlying memory technology. Consequently, heterogeneous memory systems will often have non-uniform memory access (NUMA) properties for both latency and bandwidth. To ensure optimal performance, application developers will rely on both initial page placement policies and follow-up page migration policies to ensure that hot data pages remain within the highest bandwidth or lowest latency memory node during an application's runtime [34, 75, 113].

3.2.1 Page Management Policies and Mechanisms

Modern OSes are already NUMA-aware [31, 37, 83]. In fact, standards are already being introduced that allow heterogeneous memory properties (latency, bandwidth, durability) to be exposed to the operating system so that page location decisions can be optimized to maximize throughput [152, 165]. To explain holistic OS support for multi-level memory we must conceptually separate this memory management into two distinct components. Figure 3.4 shows that memory management *policy* decisions may be driven by the kernel, device drivers, programmer, or a system administrator. These policies are all built on top of a common *mechanism*, page migration, which performs the desired OS page movement operations. Policy decisions are thus separated from page migration mechanisms through a system call (or analogous kernel interface), such as `move_pages()` on Linux.

Figure 3.4 also shows that a generic page migration mechanism involves ① allocating a new page, ② unmapping the existing virtual to physical address translation (and, on many architectures, issuing a TLB shootdown), ③ copying data from the old physical page into the new physical page, ④ mapping the virtual address to the new physical page, and finally ⑤ freeing the old physical page. The actual copy between the old and new physical page occurs only in step ③. Steps ①–② and ④–⑤ are overheads needed to ensure both correctness and protection guarantees so that neither the old page nor the new page is accessed during the page migration process. This work investigates the natural evolution of this multi-step process to leverage modern hardware and improve migration throughput.

Today, Linux uses autoNUMA [34] to try and fairly balance memory and compute requirements between NUMA nodes. It relies on two basic techniques to do this: process migration and page migration. Unfortunately, process migration is not applicable to to heterogeneous memory systems, where multiple memory nodes are all connected to a single processor. Page migration is also currently limited in multi-level memory systems because autoNUMA can only migrate pages to a memory node with free space; otherwise, pages are swapped out to disk. This is at odds with the system's goal of

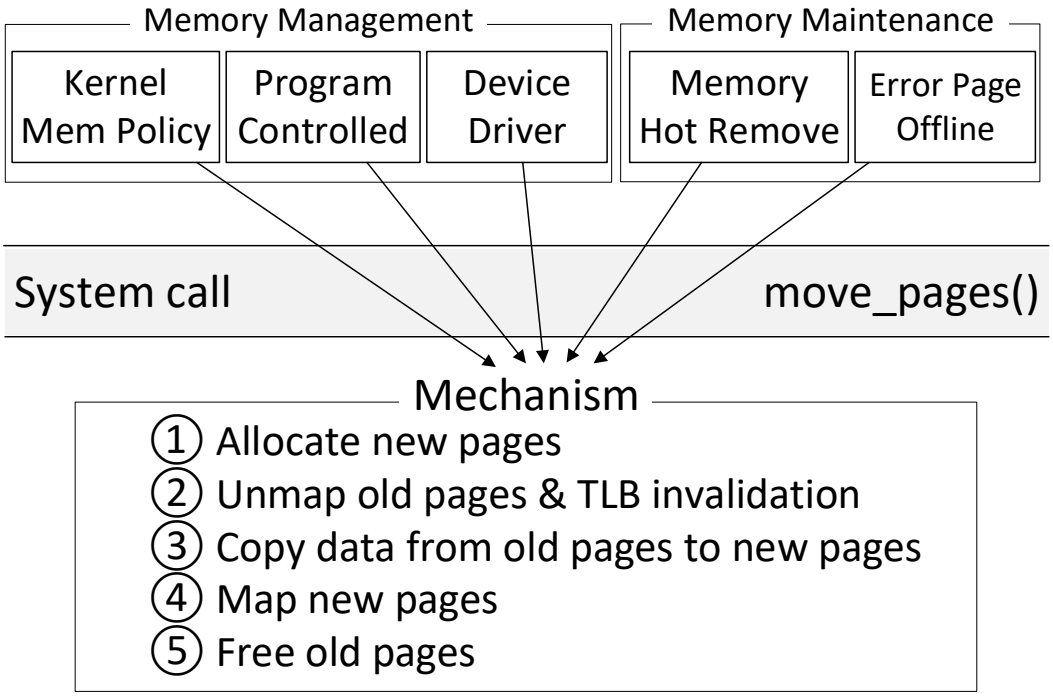


Figure 3.4: Separation of page migration policy and page migration mechanism in a multi-level memory system.

placing as many pages in a (small) fast memory as possible. Additionally, to obtain page access information autoNUMA offlines pages for profiling, and such offlining causes unpredictable memory access latency and bandwidth. These two issues have spurred a range of academic work on two-level memories; however, prior studies have generally focused on page migration policy while assuming page migration mechanisms should be sufficient [4, 28, 38, 40, 58, 85, 101, 116, 130, 163].

3.2.2 Recent Developments

Because heterogeneous memories are only now being adopted commercially, multi-level memory paging remains an active area of research. Researchers have begun exploring hardware techniques to identify hot/cold pages and facilitate their movement among memory devices [29, 56, 69, 93, 129, 142]. This work has spurred further studies on software-based approaches that are better able to manage heterogeneous memories with complex topologies [3,4,55,75,116,163]. While these studies have established the appeal of OS-managed multi-level memories, they mostly rely on trace-based simulation (which

excludes OS effects) [3, 103, 163], use non-standard OS plumbing (such as the use of reserved page table entry bits) [4], hoist page migration out of the OS (due to the unavailability of the source code or the prohibitively large engineering work) [26,31,111, 112, 136], and/or simply assume that the OS can deliver the full hardware bandwidth (which we demonstrate is not possible) during page migrations [3].

Page migration mechanisms themselves have not been studied as extensively as policies, despite being critical to performance in multi-level memory systems [50]. Combined, these seemingly subtle issues may hinder real-world adoption of many prior proposals because their conclusions may ultimately be shown to be incomplete. Solutions must be generally maintainable to be adopted in practice [53, 102]. However, this growing body of work does point to the need for both better page migration mechanisms and policies, in addition to system level evaluations of what effect the OS will have on multi-level memory systems, both of which we now address.

3.3 Native OS Support for Multi-Level Memories

Holistic support for multi-level memory systems includes both intelligent memory management policies and efficient page migration mechanisms. In this section we present our page migration mechanism improvements, which are independent of any one particular policy, as well as one specific low-overhead policy that we use in Section 3.4 to demonstrate the end-to-end benefits of our memory management system.

3.3.1 Optimizing Page Migration Mechanisms

Four critical issues need to be addressed within the operating system to implement an efficient page migration mechanism:

Larger data sizes: With larger data sizes, the software overheads of page migration can be amortized away.

Multiple threads: Today, page migration is single-threaded primarily for the sake of simplicity, but using multiple threads would speed up the copy time itself.

Concurrent Migrations: Performing multiple migrations concurrently can help us avoid the Amdahl's Law bottleneck seen in Figure 3.2. This problem cannot be solved by simply using larger pages, since architectures only support a very limited set of page sizes, and in general the largest page sizes (e.g., 1GB on x86) are not supported transparently.

Efficient two-sided migration: When a page is migrated to fast memory, a victim page must be migrated to slow memory to make room. By eliding allocation/deallocation and simply exchanging pages, two-sided operations can be faster than the sum of two one-directional migrations.

We address each of the above issues in turn below.

Native THP Migration

Our first optimization is to implement native THP migration. THP migration decreases both the hardware and software overhead of migrating THPs by a factor of 512, due to not splitting pages and reducing the number of required TLB invalidations and shootdowns. It also improves the amount of data migrated within a single migration operation. Although it may appear obvious, page migration support for THPs in Linux (and many OSes) is not yet mature, general, or high-performance. Page migration was originally proposed to enhance NUMA system performance and achieve memory hotplug functionally before THPs had even been introduced [7, 82].

Today, for example, Linux cannot migrate THPs in response to programmatic resource management requests like *mbind* and *move_pages*, which are designed to move data to specific memory nodes at the request of programmers. This is a serious shortcoming for heterogeneous memories since there are important situations where programmer-directed data placement enables good performance [116]. Similarly, Linux cannot directly migrate THPs in response to memory hot removal, soft off-lining, or *cpuset/cgroup*. In all these cases, when Linux is migrating a virtual memory range that contains transparent huge pages, it must split the THPs and migrate the constituent base pages instead, resulting in poor page migration performance and reduced TLB

coverage.

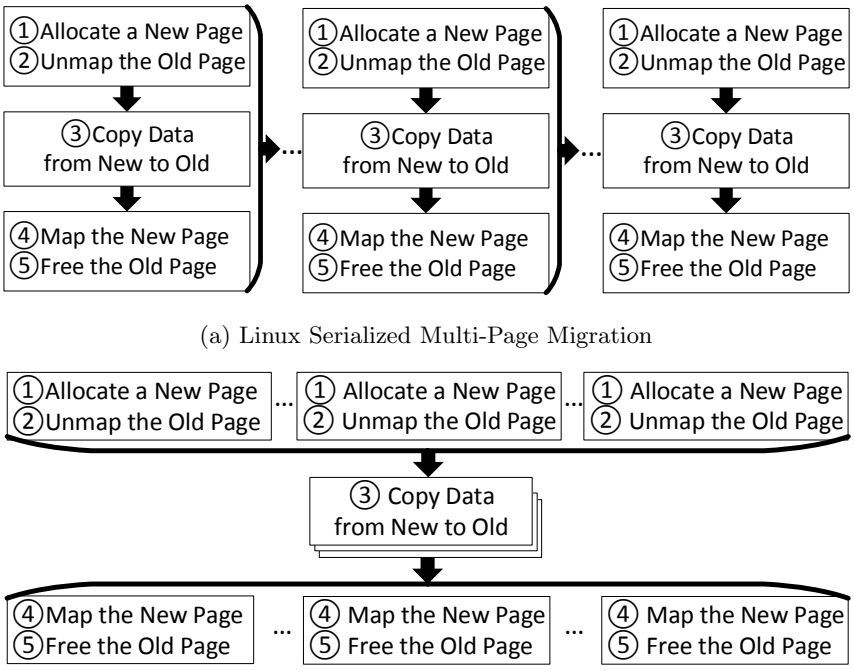
To implement THP migration, we augment the five steps shown in Figure 3.4 to be THP aware. Our THP implementation supports *all* resource management requests (i.e., *mbind*, *cpuset*, etc.) In addition, we adjust other THP-specific code paths in the kernel to account for the fact that THPs may be undergoing migration at the time of the call. We do this either by waiting for the end of the migration or by simply skipping the THPs (as is done for base pages). Other OSes can follow the same principle to realize THP migration to enable significantly higher-throughput page migration for a better support of heterogeneous memory systems.

Parallelized THP Migration

Currently, the Linux kernel page migration routine is single-threaded and has a limited amount of data (usually the size of one base page) to transfer within a single migration operation. Motivated by Figure 3.3, we implemented a variable-thread count based copy subroutine within the page migration operation.

To enable multi-threaded page copies within the Linux `move_pages()` system call, we use kernel `workqueues` to spawn helper threads to copy data between arbitrary physical ranges. Our implementation calculates the amount of data to be copied via each parallel thread by dividing the page size (or, in Section 3.3.1, the aggregate of pages being migrated concurrently) by the selected number of worker threads.

Because the exact selection of thread location and thread count needed to maximize throughput is likely to differ among systems, we provide parameter configuration through the `sysfs` interface, so that system administrators can enable or disable multi-threaded CPU copy or change the number of CPUs involved in the data copy. We also augment the `move_pages()` system call with an optional parameter flag, `MPOL_MF_MT`, to enable migration policy engines to dynamically choose the level of parallelism on a per migration basis.



(a) Linux Serialized Multi-Page Migration

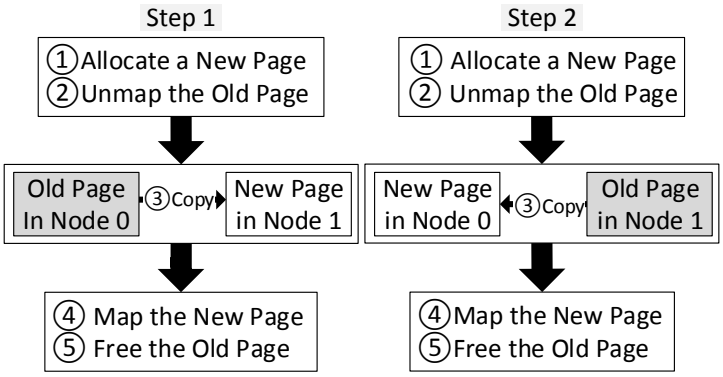
(b) Proposed Concurrent Multi-Page Migration

Figure 3.5: Improvements to Linux multi-page migration to enable large transfers for improved copy bandwidth.

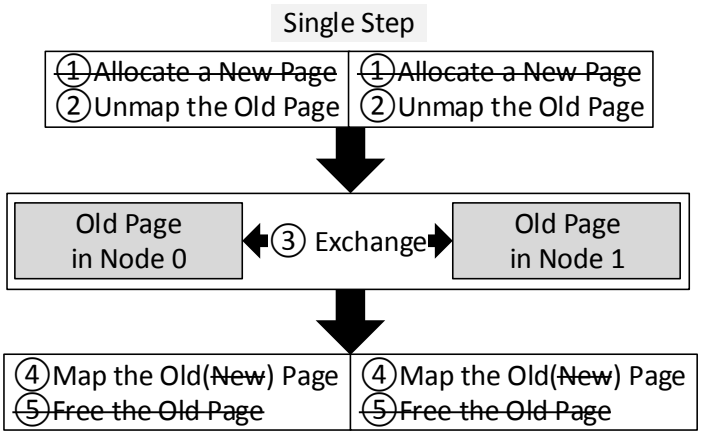
Concurrent Multi-page Migration

Multi-page migration is expected to be common in multi-level memory systems due to spacial locality and prefetching effects. The Linux `move_pages()` interface already supports migration of multiple pages with a single system call by passing in a list of pointers to the pages to be migrated between memory nodes. However, as shown in Figure 3.5a, the current implementation serializes the copies and performs them one page at a time.

Our new page migration implementation concurrently migrates all pages in the list provided to `move_pages()` by aggregating all data copy procedures into a single larger logical step, as shown in Figure 3.5b. As an example, consider the case of migrating 16 THPs of 2MB. In the current Linux implementation, even using the parallel copy optimization, Linux will transfer 16 THPs of 2MB, with an implicit barrier between each parallel 2MB copy. In our concurrent migration optimization, each of the pages in the list is allocated and assigned a new page with matching size and then unmapped.



(a) Exchange Two Page Lists in Linux



(b) Proposed Exchange Page Migration

Figure 3.6: Exchanging pages improves efficiency by eliminating memory allocation and release when migrating symmetric page lists between memory nodes.

Then, all pages in the list are distributed to the per-CPU `workqueues` according to the `sysfs` configuration. If more parallel transfer threads are available than concurrent pages to migrate, our implementation uses multiple threads to copy different parts of a single page to maximize throughput. Once the concurrent page copy step is completed, the new pages are mapped onto the correct page table entries and the old pages are freed. It may also be possible to parallelize other steps of page migration. However, because there are strict correctness requirements for synchronization including architecture dependent page table manipulation, complicated situations arise involving failure recovery if optimizations become too aggressive.

Symmetric Exchange of Pages

In multi-level memory systems, single-ended page migration is unlikely to be the common case. Higher-bandwidth memory is generally capacity-limited compared to larger lower-bandwidth memories. Therefore, when migrating pages into a higher-level memory node, at steady state, the page migration policy will need to migrate pages out of that node, so as to not exceed physical memory capacity. Therefore, when managing a high performance heterogeneous memory node as a software controlled cache, each hot-page insertion requires a symmetric cold-page eviction.

Naive two-step, one-way migration makes inefficient use of system hardware if the two copies are protected by locks and executed serially, as is the case in today's OSes. Figure 3.6a shows this common two-step page migration operation. First, the locking serialization limits the benefits of our previously discussed optimizations. Second, each migration operation must perform independent physical page allocation and deallocation, and both are expensive software overheads (as shown in Figure 3.2).

To eliminate these overheads, we propose to combine the two one-way migration operations into a new single symmetric exchange operation. By exchanging pages, our implementation eliminates many of the kernel operations required in unidirectional page migration and reuses the existing physical pages instead of allocating new ones (as shown in Figure 3.6b).

We implement symmetric page migration in Linux by providing a new `exchange_pages()` system call that accepts two equal-sized lists, of equal-sized pages. If the page lists do not meet the requirement, the caller must revert to a traditional two-step migration process. When called with two symmetric page lists, our exchange of pages implementation follows a similar path to unidirectional page migration. The differences are that that no new pages are allocated, and that instead of copying data into new pages, we transfer data between each pair of pages using copy thread(s) that use CPU registers as the temporary storage for in-flight iterative data exchange operations. This use of registers allows our mechanism to avoid allocating a complete temporary page.

Both our parallel page copy and concurrent page migration optimizations focus on

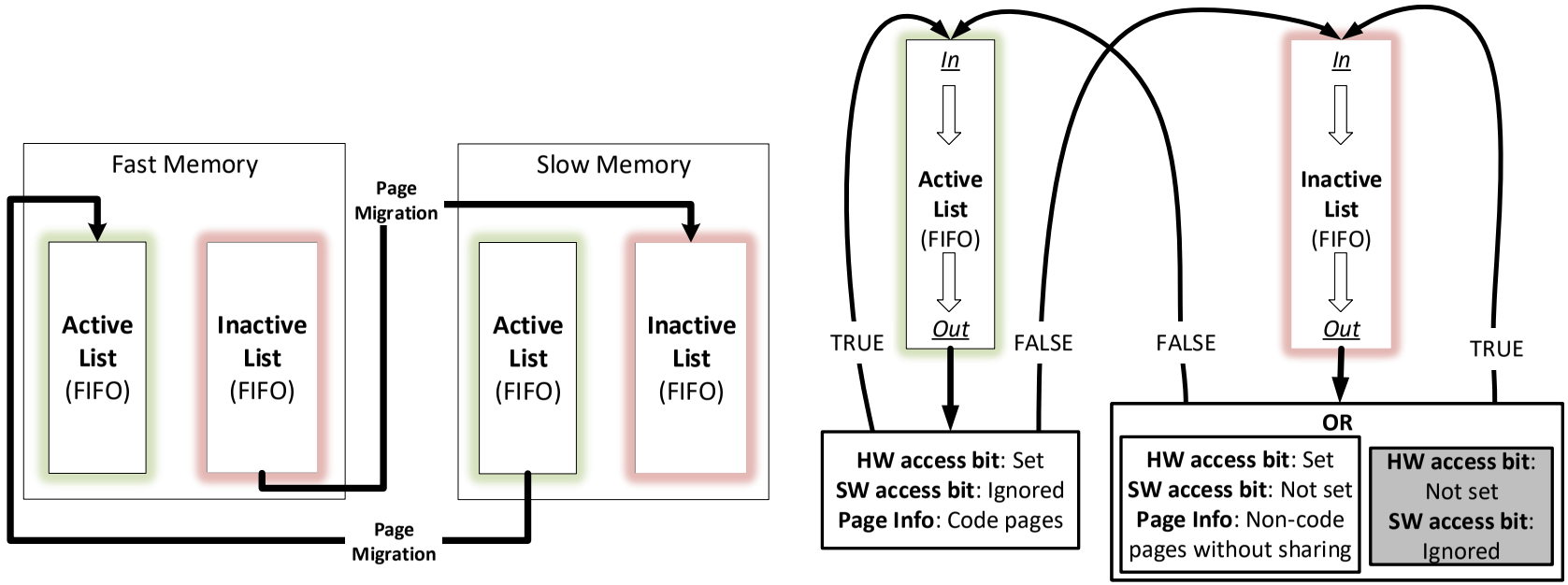
improving the data copy itself while symmetric page exchange eliminates two of the most expensive software overheads that occur during page migration: page allocation and release. Because these kernel operations consume constant time irrespective of page size, page exchange improves the migration throughput of both base pages and transparent huge pages if memory management policies choose to use it. Additionally, pairs of pages can also be exchanged using parallel exchange (Section 3.3.1), concurrent exchange (Section 3.3.1), or both, without extra locking as long as each parallel exchange thread operates in isolation.

3.3.2 Optimizing Page Tracking and Policy Decisions

A multi-level memory paging policy and system needs to be sufficiently general and representative of real-world scenarios in order to be broadly useful on the diverse set of heterogeneous memory systems that are beginning to emerge. To this end, we propose a page migration policy that is simple, has been shown to work well in a wide range of environments [75], and adds negligible overhead to the baseline. As such, our implementation builds upon the existing Linux page replacement algorithm. We intentionally make as few changes as possible to keep our implementation maximally compatible with the upstream kernel.

The goal of a page replacement algorithm is to identify hot and cold pages so that the page migration mechanism can migrate hot pages out of (historically) disk or (in our case) slow memory, and into fast memory. Linux already achieves this by separating hot and cold pages within each memory node into *active* and *inactive* lists, where a page can be in only one of these lists at a time. As hot pages become cold and vice versa, the kernel actively moves the pages between these lists as shown in Figure 3.7.

Like Linux's, our policy moves pages from one list to another by checking each page's state and two access bits, one in the page table entry pointing to the page and the other in the metadata of the page maintained by the kernel. We call the former the hardware access bit and the latter the software access bit. A page table entry's access bit is set by the hardware page table walkers on the first TLB miss to each virtual-to-physical translation corresponding to that page. Its software access bit is set by the



(a) Inter-memory-node page migration

(b) Intra-memory-node page list manipulation

Figure 3.7: Proposed native multi-level paging policy consisting of (a) inter-memory node page migration and (b) intra-memory-node page list manipulation. The former migrates hot pages (in the *active* list) from slow to fast memory and vice versa for cold pages (in the *inactive* list). The latter moves pages within a given memory node from one tracking list to the other.

existing Linux paging algorithm for each physical page. Both hardware and software access bits are inspected using the atomic operation `test_and_clear()`, except where marked "*Ignored*" in Figure 3.7, in which case the bit is neither checked nor cleared.

The key difference between our proposed paging policy and the standard Linux approach is graphically illustrated with the greyed box in Figure 3.7. In Linux today, cold pages that have not been accessed recently can be reclaimed (freed or paged to disk). However, heterogeneous memory systems aim to percolate such pages to the slower memory instead, to avoid the high cost of paging them back in from disk later. Consequently, our policy chooses to keep this page in the inactive list to make it a candidate for migration out of the fast memory. Similarly, if capacity is available in our fast memory (e.g., due to memory deallocations or migration of inactive pages), pages from the slow memory active list will be migrated into the fast memory. If the fast memory is full and does not contain inactive pages, no migration will occur.

Similar to traditional NUMA allocation policies, when new memory is allocated, it will occur in the fast memory if free space is available; otherwise, it will only occur in the slower memory node. We do not evict pages upon allocation so that we can keep page migration off the memory allocation path, which is performance critical. Finally, our system optimizes page locations only every 5 seconds throughout application runtime to minimize active process interference, based on profiling results.

3.4 Experimental Results

To quantify the utility of optimized page migration in a concrete scenario, we evaluate our approach on a disaggregated memory system due to the emerging importance of these systems in industry [55, 87].

3.4.1 Methodology

We emulate a disaggregated memory system using an experimental machine that has two memory nodes; we use one as fast local memory and the other as slow remote memory. We emulate slow memory by running one or more instances of memhog, an

Intel Xeon Dual Socket System	
Processors	2-socket E5–2650v3
Memory	DDR4 — 2133MHz
Cross-socket QPI BW	19.2 GB/s
Memory BW	34.0 GB/s (per-socket)
Memory Latency	84.9 ns
OS & Kernel	Debian Buster — v4.14.0
Disaggr Mem BW (Emulated)	17.0 GB/s
Disaggr Mem Latency (Emulated)	199.2 ns

Table 3.1: Overview of experimental system.

artificial memory-intensive workload that has been used in prior studies to load the system [116, 124]. The memhog instances, which run on otherwise-idle CPUs, inject extra memory traffic into the system. This has the effect of reducing remote memory bandwidth to one half of local memory bandwidth and increasing unloaded access latency to double that of local memory, which has been validated by Intel Memory Latency Checker [63]. Table 3.1 gives additional details of our setup.

To evaluate our OS optimizations, we integrate our proposed optimizations into Linux v4.14. The statistics of kernel modification given by `git diff` is: `23 files changed, 627 insertions(+), 114 deletions(-)`. We intend to open-source our code upon publication.

For the evaluation itself, we perform a variety of experiments. First, we evaluate a set of microbenchmarks to measure the effect of each of our proposed optimizations in isolation and combination. Second, to get complete end-to-end performance numbers, we run workloads from SpecACCEL [72] and graph500 [108], and we show the performance across a range of fast memory oversubscription scenarios. Third, we sweep the design space to highlight the interesting behaviors that arise and to identify the configuration parameters that perform the best. Finally, we evaluate them on additional non-x86 architectures to prove the generality of our proposed enhancements.

3.4.2 Page Migration System Call Performance

To build intuition as to the sources of the overall performance improvements achieved by our page migration mechanisms, we first use microbenchmarks to tease out the

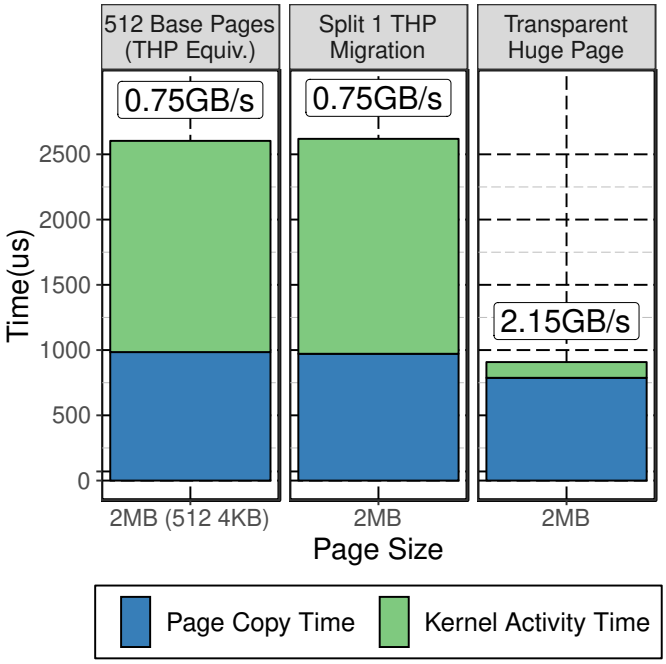


Figure 3.8: Cost breakdown of 512-base-page migration, THP-split migration, native THP migration.

relative benefits of the different (complementary) optimizations. Our experiments use the generic page migration interfaces, `move_pages()` (with our optimizations), and `exchange_pages()` (our newly proposed system call).

Native THP Migration

Figure 3.8 contrasts the performance of a kernel that migrates pages in three ways. The leftmost bar is unmodified Linux migrating 512 4KB pages. The middle bar is Linux migrating a 2MB THP by splitting it into 4KB pages, which are then migrated. The right bar is our proposed native THP migration support.

Migrating a 2MB THP with splitting achieves virtually identical throughput as migrating 512 4KB pages. This is unsurprising, since they perform the same kernel operations for 512 pages plus one additional THP split. Our native THP migration improves throughput by 2.9× because it reduces kernel overhead by consolidating 512-page operations into a single page operation. Figure 3.8 also shows that page copy time decreases only marginally; i.e., there is still significant room for our additional optimizations to improve overall throughput.

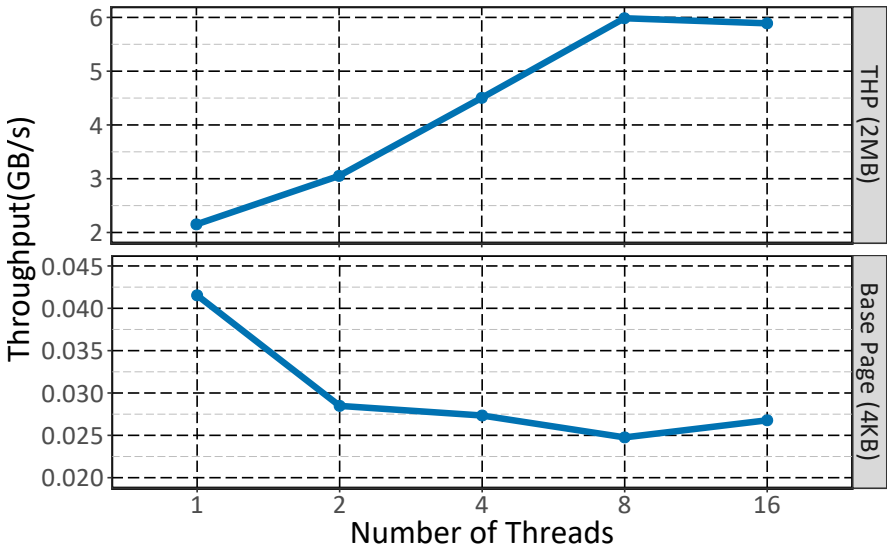


Figure 3.9: Throughput (**higher is better**) of multi-threaded single page migration for both base page (4KB) and THP (2MB).

Multi-threaded Transfers

Figure 3.9 shows the results of our first optimization, parallel (multi-threaded) page copying. We separate results for the cases where we migrate 2MB THPs (the graph on the top) and 4KB base pages (the graph on the bottom) also varying the number of threads used to perform the copies.

There are two primary observations from our multi-threaded copy results. First, parallel page copies are primarily beneficial when the page sizes are larger. For example, 2MB THP page migration time is sped up 2.8×, while parallel copies do not improve the throughput for 4KB pages, because the thread launch overhead cannot be amortized sufficiently. This overhead is likely the reason that the current Linux page migration has remained single-threaded regardless of base page sizes. Second, the graph at the top of Figure 3.9 shows that even though parallel migration is useful for 2MB THPs, overall throughput still remains well below the maximum cross-socket copy throughput of around 16 GB/s (see Figure 3.3), motivating the need for concurrent page migrations.

Concurrent Page Transfers

Figure 3.10 illustrates the performance advantage of using concurrent page migration optimization whenever possible. Results are again separated for 2MB THPs (the graph on the top) and 4KB base pages. For 4KB pages, parallel non-concurrent migration is always inferior to single threaded migration (due to the aforementioned parallelization overheads), and concurrent parallel migration only surpasses the baseline at sufficiently large page counts. As previously mentioned, the OS overheads are too large to overcome. However, utilizing both parallelism and concurrency are clear wins when transferring 2MB THPs, with a performance advantage (over parallelism alone) ranging from 10-25% depending on the number of pages transferred.

Symmetric Exchange Pages

Figure 3.11 shows the benefits of symmetric page exchange atop our prior optimizations. When using THPs, exchange page throughput follows similar trends as concurrent page migration, but with a performance improvement ranging from 10-50% depending on the number of pages exchanged. Interestingly, the largest fractional improvement is when exchanging small numbers of pages, because in these cases the software overhead remains a significant fraction of total transfer time. Removing the memory management overheads from the page migration process can improve the throughput of page migration to as high as 11.2GB/s when exchanging two lists of 512 2MB pages (1GB data on each list); this is very close to the best achievable copy throughput (excluding kernel overhead) of 11.7GB/s.

Unlike our prior optimizations, when exchanging base pages (4KB) we also observe throughput improvements. When exchanging two lists of 512 4KB pages (2MB data on each list), we get 1.1GB/s throughput, or 37.5% more than the throughput of Linux's base page migration.

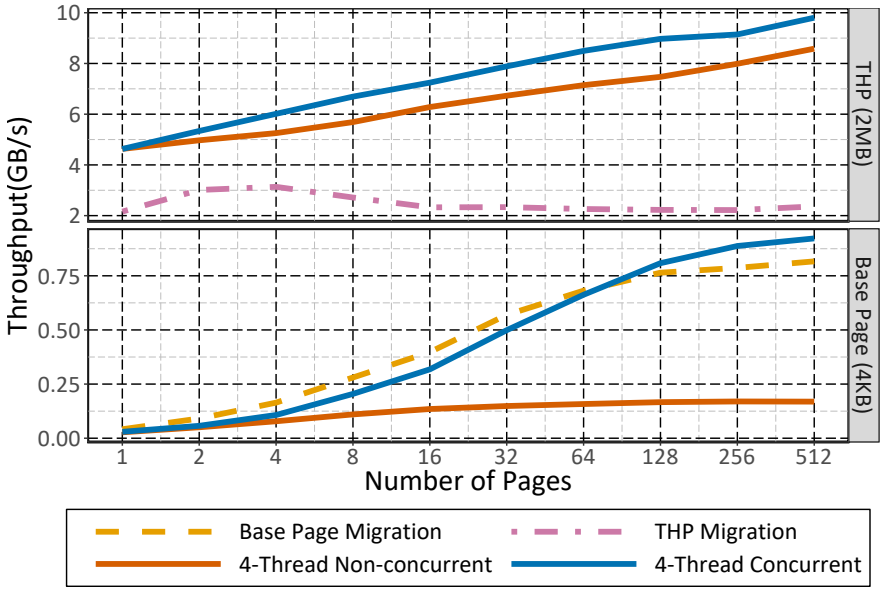


Figure 3.10: Throughput (**higher is better**) of concurrent page migration for both base page (4KB) and THP (2MB) with different numbers of pages under migration. 4-Thread Non-concurrent uses 4-thread data copy and 4-thread Concurrent adds concurrent page migration. Single-threaded Base Page Migration and THP Migration are shown for reference.

Microbenchmark Summary

When using only base pages and the three non-THP improvements (multi-threaded copy, concurrent copy, and two-way exchange), our system yields a 1.4× throughput improvement as compared to Linux's single threaded implementation. For THP migration, native THP migration alone (i.e., without our parallel/concurrent/exchange optimizations) delivers a 2.9× migration throughput improvement over Linux's state of the art. With our parallel copy and concurrent page migration optimizations added, we achieve a 4.6× throughput improvement over native-THP migration. With two-way exchange, we further improve throughput by 1.1×. The combined overall improvement for THP migration over the Linux baseline is 5.2× versus THP-splitting migration and 15× over base page only migration.

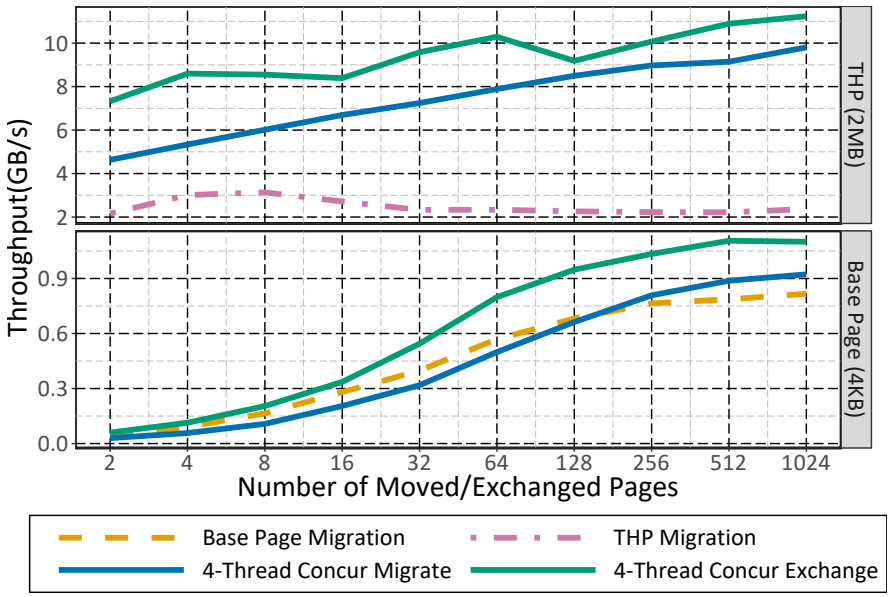


Figure 3.11: Throughput (**higher is better**) of page exchange vs. 2 page migrations for both base page (4KB) and THP (2MB) sizes while varying the number of pages exchanged. 4-Thread Concur Migrate and 4-Thread Concur Exchange use both concurrent and 4-thread parallel data copy. Single-threaded Base Page and THP Migration throughput are shown for reference.

3.4.3 End-to-End Performance Results

Thus far, we have used microbenchmarks to quantify the performance benefits of our page mechanism optimizations. We now turn our attention to evaluating the end-to-end performance improvements that these optimizations, combined with our low overhead page management policy, can achieve. Our experimental testbed emulating a disaggregated memory is described in Section 3.4.1. No changes have been made to Linux's THP allocation policy and they are provided, when possible, on demand by the operating system.

We evaluate SpecACCEL and graph500, with the memory footprints scaled to 32GB. With these workloads running under our experimental setup, we find that over 90% of the pages for each workloads is typically backed by THPs, indicating that there is a significant negative impact of having to split THPs into base pages before migration.

We first run each workload in a disaggregated memory scenario that has 16GB local memory and 40GB remote memory. In this configuration, the local memory is only half the size of the workload memory footprint, while the remote memory can accommodate

the entire workload footprint. We compare four different page migration mechanisms, along with an upper and lower bound for comparison:

1. **All Remote**, the lower bound, where workloads are run entirely from the 40GB remote memory
2. **Base Page Migration**, the Linux default (THPs are split before migration)
3. **Opt. Exchange Base Pages**, 4-threaded parallel copy and 512-page concurrent exchange (THPs are split before migration)
4. **THP Migration**, our native THP approach without parallel, concurrent, or exchange optimizations
5. **Opt. Exchange Pages**, THP migration, 4-threaded parallel copy and 8-page concurrent exchange
6. **All Local**, the upper bound, where workloads are run entirely from a 40GB fast local memory

In configurations 3 and 5, we use 4 threads for copying and 512 and 8 pages respectively for our migration parameters. From our microbenchmark results, this presents the best configuration for both base and THP migrations. In Section 3.4.5 we present further sensitivity analysis to justify these selections.

Figure 3.12 shows the relative speedup of these six configurations over All Remote. All Local, which is our ideal case, on average achieves about 2× geomean speedup over All Remote, reflecting the local vs remote memory bandwidth and access latency difference in our system. Base Page Migration, which is our baseline for managing disaggregated memories, improves workload performance on average by 9%. However, some workloads (e.g. 551.ppalm, 556.psp, and graph500) perform worse than All Remote, meaning Base Page Migration is not always making good use of the 16GB local memory. Opt. Exchange Base Pages improves workload performance by 16% on average, and in this case, only graph500 does not take advantage of the 16GBs of local memory. Both of these results are testament to the notion that poor migration mechanisms can actually *degrade* performance despite the addition of a faster tier of memory.

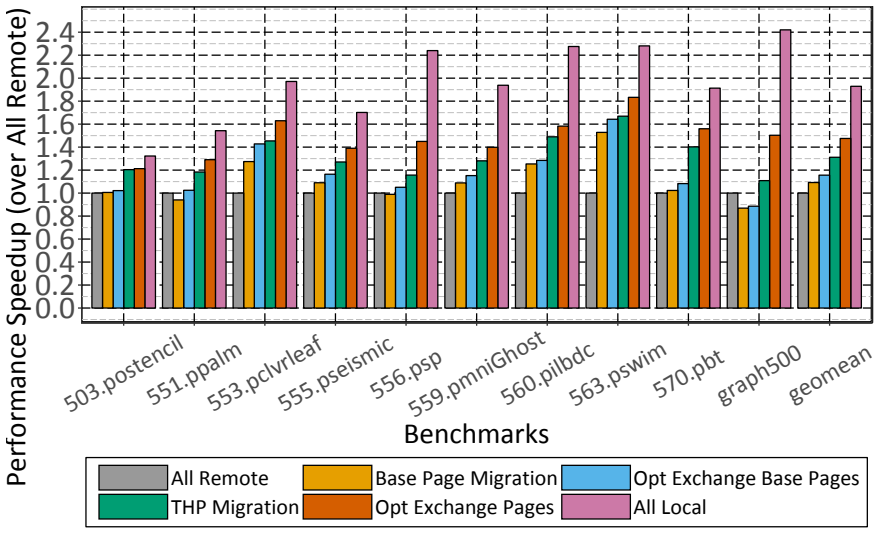


Figure 3.12: Benchmark runtime speedup (over All Remote) with 16GB local memory. Base page migration and THP migration are single-threaded and serialized and shown for comparison, while Opt Exchange Base Pages uses 4-thread parallel and 512-page concurrent migration and Opt Exchange Pages use 4-thread parallel and 8-page concurrent migration.

Fortunately, enabling our native THP Migration can indeed harness the benefit of fast memory. Enabling THP migration improves the geomean workload performance by 31% over All Remote and achieving 68% of the geomean All Local performance, which is our ideal case. Our Opt. Exchange Pages (which can migrate both base pages and THPs) further improves average performance by 48% over All Remote and achieves 77% of the geomean ideal All Local performance.

3.4.4 Sensitivity to Local Memory Size

To further demonstrate the general applicability of our approach, we sweep the local memory size from 4GB to 28GB and show the geomean of all benchmark speedup over All Remote. Figure 3.13 show that the performance trends are similar to those of the 16GB local memory case and we note four key observations First, Linux's Base Page Migration is not able to exploit the full potential of the disaggregated memories. When the local memory size (such as 4GB and 8GB) is much smaller than the workload's 32GB memory footprint, it degrades workload performance by 5% to 10% on average. Second, our Opt. Exchange Base Pages can help improve performance but is still limited

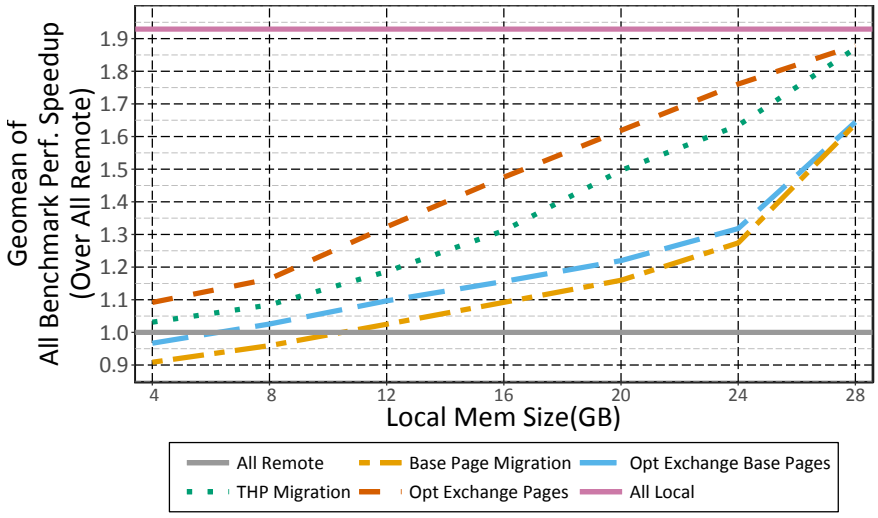


Figure 3.13: Geomean speedup (over All Remote) over a sweep of local memory sizes, from 4GB to 28GB. Base page migration and THP migration are single-threaded and serialized, while Opt. Exchange Base Pages uses 4-thread parallel and 512-page concurrent migration, and Opt. Exchange Pages uses 4-thread parallel and 8-page concurrent migration.

by page migration throughput. It is however slightly better than Base Page Migration. Third, our THP Migration keeps improving performance; thus, it is able to make using disaggregated memory feasible without any performance loss. Fourth, our Opt. Exchange Pages improves performance substantially, exploiting most of the potential in the disaggregated memory system and outperform Linux's Base Page Migration 40% on average.

3.4.5 Sensitivity to Tunable Parameters

Our page management system provides a number of user-tunable parameters. Here, we evaluate how sensitive the performance is to two of those parameters: the parallel copy thread count and the number of pages migrated concurrently.

Number of threads used for parallel page migration

Because our system allows the number of threads used to copy data to vary, it is important to understand the importance of this tunable parameter. Using higher thread counts will improve copy throughput, but steals compute resources from the application

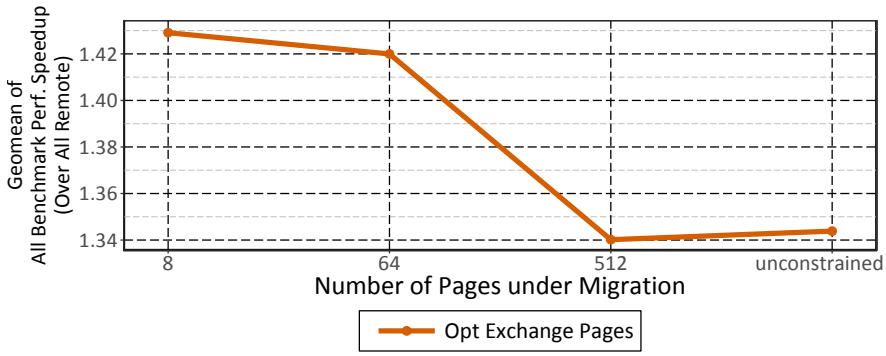


Figure 3.14: Geomean speedup (over All Remote) given different numbers of pages under migration. Opt Exchange Pages use 4-thread parallel data copy. "*Unconstrained*" means we do not limit the number of pages under migration; instead we just migrate all pages at once. All use 16GB Local Memory.

itself, thus how to strike a balance between these factors is important to understand.

We swept the number of threads used by parallel page migration from 1 to 16. We discovered that using 4 copy threads was almost always the highest performing configuration, but that performance when using other thread counts had a maximum variance of just 4%, indicating that our proposed system is not overly sensitive to this parameter. Therefore, simply selecting a reasonable point (such as the 4 thread configuration we used in our end-to-end results) is a reasonable decision.

Number of pages being migrated concurrently

Similar to the variation possible in copy threads, we tested our system's sensitivity to the number of pages we allow to be migrated concurrently. As the number of migrated pages increases, so should throughput; however, beyond a certain point, migrating a high number of pages also means that these pages are inaccessible (because they are in-flight and unavailable to the user process). This can result in application stalls and decreased performance.

Figure 3.14 shows the effect of varying the concurrent migration page count. We observe that limiting the number of concurrently migrated pages in the system is necessary, with 8 pages (32KB or 16MB of data respectively depending on page size) performing best. There is very little performance variance when using any value less than 64 pages, but performance does drop by almost 10% if the number is left unconstrained.

Platform	Page Migration Type			
	Base Page	**THP**	**Opt.**	**Opt. Exch.**
Intel Xeon	0.75	2.15	9.80	11.23
IBM Power 8	1.24	8.70	23.20	26.88
NVIDIA TX1	0.64	1.36	-	-

Table 3.2: Maximum achieved huge page migration (in GB/s) throughput based on architecture independent optimizations (bolded) shown across three architectures. When on NVIDIA TX1 (ARM64), due to platform constraints, we are only able to run THP migration.

3.4.6 Architectural Independence of OS Optimizations

To explore the platform independence of our multi-level memory system and optimizations, we port them to other hardware platforms and compare the microbenchmarks used in Section 3.4.2. Our multi-level memory page tracking and policy implementation is architecture independent by design, because it is based on Linux's current architecture-independent active and inactive page list implementation. Thus we focus on platform sensitivity for our migration optimizations. We show results in Table 3.2 using optimal tunable parameters, as the best configuration for each platform varies due to available memory bandwidth and CPU performance.

While all platforms benefit from our optimizations, using Intel Xeons, we get a 2.9× improvement in migration bandwidth using native THP migration, an additional 4.6× by employing parallelization and concurrency optimizations, and another 1.1× by utilizing our new `exchange_page()` interface, which results in 15× total improvement. For Power, we achieve 7.0×, 2.7×,and 1.2× the throughput, respectively, with the same optimizations. This results in a 21.7× page migration throughput improvement. For an NVIDIA TX1 (ARM64), we observe 2.1× throughput with THP migration as compared to base page migration. However due to lack of NUMA support on this hardware platform, we do not include concurrent THP migration and exchange of pages, as measuring results within a single socket may skew the results.

Additionally, on the Xeon platform there is variance in copy throughput depending on the x86_64 data copy instructions (integer vs. floating-point) used. However our testing finds that SIMD floating-point instructions, like SSE and AVX, no longer provide significantly higher copy throughput than `mov` on x86_64 due to aggressive linear

prefetching within caches for easily identified memory patterns like page migrations.

On Power systems, a single-threaded data transfer can achieve almost 10GB/s of copy bandwidth regardless of transfer size, but this is still less than 50% of the maximum achievable copy bandwidth when using multiple threads. The improved single-threaded throughput on Power arises from the use of an integer vector move instruction that moves data at an efficient 16-byte granularity. CPU instructions and architectures that can improve single threaded copy bandwidth will ultimately help both base and transparent huge page migration, but it is unlikely that even vector instructions can achieve the 15× improvement needed to match the aggregate performance this work achieves through software only techniques.

3.4.7 Summary of Experimental Results

To summarize, in heterogeneous memory systems such as the disaggregated memory system we use, low-throughput page migration mechanisms (such as Linux's baseline) are not able to exploit the benefits of fast memory. In fact, the existing mechanisms often even degrade performance to the point that they may result in runtime that are worse than simply running on a system without any fast memory.

Fully releasing the potential of multi-tiered memory requires enabling our four key optimizations: THP migration, multi-threaded copy, concurrent migration of multiple pages, and two-way exchange. Our evaluation shows that the heterogeneous memory page management system that we propose is able to deliver significant speedup across multiple benchmarks, and our design space exploration shows that our techniques are flexible and general enough to apply across a range of architectures and memory system configurations.

3.5 Related Work

Heterogeneous Memory: Hybrid memory systems consisting of high-bandwidth, low-capacity memory (e.g. Hybrid Memory Cube (HMC) [105, 122], High Bandwidth Memory (HBM) [68]) and low-bandwidth, high-capacity memory (e.g. DDR, NVM [98])

are being widely adopted by vendors [65,114]. Recent work has investigated architecting high performance memory in a hybrid memory system either as a hardware-managed cache [129, 143] or part of the OS-visible main memory system [29, 65, 106, 142]. When hybrid memory is OS-managed, performance is primarily dependent on the service rate from the high bandwidth, low capacity memory.

Page Placement Policies: To effectively utilize hybrid memory system performance, prior art focuses on page placement *policies*. Such policies use heuristics or hardware counters for page access profiling [131, 150], dynamic page access tracking [103], or bandwidth partitioning [3]. These page access profiling techniques either require specialized hardware (precluding policy portability) or incur high overhead, reducing the number of profiled pages. For example, consider that autoNUMA is carefully designed to balance the costs of profiling with the benefits of *accurate* profiling. Since autoNUMA uses page faults to sample data (which can consume $\sim$ 1000 cycles [151]), it limits its sampling rate to reduce the performance problems of excessive page faults. Thermostat [4] samples page hotness using page faults (via BadgerTrap [47]). This can cause $\sim 4\times$ slowdown if all pages are profiled; consequently, Thermostat only profiles 0.5% of total memory.

HeteroOS [75] targets heterogeneous memory in virtualized environments. This work adapts Linux's page replacement algorithm as we do, but relies on page hotness tracking. This tracking mechanism can cause frequent and expensive TLB invalidations; consequently the authors are careful to limit aspects of their tracking mechanism. Like our work, HeteroOS shows that the high overhead of page migration makes heterogeneous memory system management suboptimal but does not address the issue further.

Page Migration Mechanisms: Similar in spirit to optimizing page migration, one recent study focuses on enhancements to the DRAM architecture to migrate pages within the memory controller without bringing data on-chip [137, 156]. This avoids cache pollution effects. Further, to avoid locking down page accesses during migration, other proposals pin data in caches, enabling pages to be accessed during migration [23]. Others have attempted to avoid the use of the in-kernel locks by freezing the related applications instead [86]. Lin et al. propose an asynchronous OS interface called *memif*,

for accelerating page migration with DMA devices [89]. Instead of using traditional Linux migration interfaces, it requires rewriting programs to adopt a new interface and is limited to only ARM processors due to an architectural dependence for race-detection. More recently, Ryoo et al. discover that migrating larger numbers of pages (64KB or 2MB in groups of 4KB pages) could improve application performance in heterogeneous memory systems based on their leading-load model, which further supports our THP migration proposal [136].

Huge Page Management: Ingens [81] is a huge page management framework. Their focus in on principled ways to coordinate the construction of more THPs. Our work asks a complementary question: how can we preserve THPs? Our work can speed up their huge page promotion process and reduce memory fragmentation by preserving THPs during migrations.

3.6 Conclusions

Current OS page migration and management frameworks were developed at at time when page sizes were small and page migration existed to support memory hotplug functionality rather than performance optimization. With the introduction of heterogeneous memory systems, pressure is being put on the OS to adapt to new hardware paradigms and efficiently support multi-tiered memory systems. This work implements a novel holistic high-performance page migration system which increases migration throughput from under 100MB/s to over 10GB/s, rendering it appropriate for a wide variety of future asymmetric memory studies. We also design a low overhead page tracking and migration policy, that significantly re-uses pre-existing internal OS structures, thus having wide applicability to a range of anticipated multi-level memory systems. Using a disaggregated memory system as an example, our page migration enhancements along with this native two-level paging system result in a 40% end-to-end improvement in workload performance. Our work demonstrates that combining simple management policies with high performance page migration is critical, yet achievable, to the future of high-performance heterogeneous memory systems.

Chapter 4

Translation Ranger: Operating System Support to Actively Produce Address Translation Contiguity

4.1 Introduction

Virtual memory (VM) provides programmers with a number of important benefits: it abstracts away the physical memory details, it provides memory protection and process isolation, and it facilitates communication between cores and/or compute units sharing the same virtual address space. As such, virtual memory is used today not just by CPUs but also by GPUs and many other accelerators [6, 61, 149]. Figure 4.1 shows an example of a system with a group of CPUs, GPUs, and accelerators, where each can access its own dedicated/shared memory and one another's memory directly via the virtual memory system.

Generally, in systems using virtual memory systems, the translation lookaside buffer (TLB) in each computation unit is desired to cover all physical memories, requiring considerable translation storage overhead [59,141]. Consequently, processor vendors are implementing increasingly large TLBs. For example, Intel has been (approximately) doubling its CPU TLB resources every generation from Sandybridge to Skylake [66], resulting in TLBs with thousands of entries today. Vendors like AMD implement even larger TLBs for accelerators like GPUs [94,153]. Large TLBs consume non-trivial area and power [14, 36, 77, 120] and are particularly ill-suited for accelerators with limited hardware resources [59, 128, 141].

Recent studies focusing on this address translation wall [17] have touted *translation contiguity* as a way to mitigate VM overheads [13, 30, 76, 121, 125]. With translation

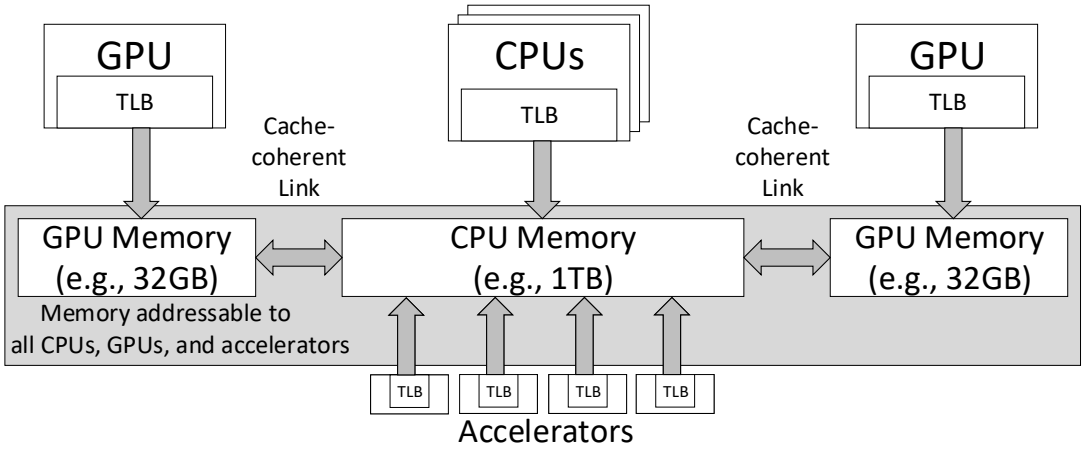


Figure 4.1: A system with CPUs and accelerators sharing memory (in a cache coherent manner). All memories are addressable by all CPUs, GPUs, and accelerators.

contiguity, virtually contiguous pages mapping onto physically contiguous frames (referred to as a *contiguous region*) can be covered by one translation entry. OSes can generate contiguity using contiguous memory allocators [104], and hardware can take advantage of contiguity via coalescing as shown in Figure 4.2 [30, 125]. Additionally, academic studies have shown how to use translation contiguity even more aggressively with recently-proposed range TLBs [76], devirtualized virtual memory [59], and direct segment hardware [13].

Although influential, prior studies require either specific amounts of contiguity (e.g., discrete page-sized contiguity [36, 119, 139]), restricted types of contiguity (e.g., where virtual and physical pages must be identity mapped [59]), serendipitously-generated contiguity where the OS offers no guarantees of contiguity creation (e.g., TLB coalescing [125]), or swaths of contiguity that can be created only at memory allocation (e.g., direct segments and ranges [13, 76]). Consequently, the question of how OSes can generate *unrestricted and general-purpose contiguity* remains open.

Our goal is to create such general-purpose contiguity: (1) during the entire lifetime of workloads and not just at memory allocation; (2) on real-world systems with long uptimes, where memory may be (heavily) fragmented by diverse co-running workloads that are spawned and terminated over time; and (3) that does not require OS/application

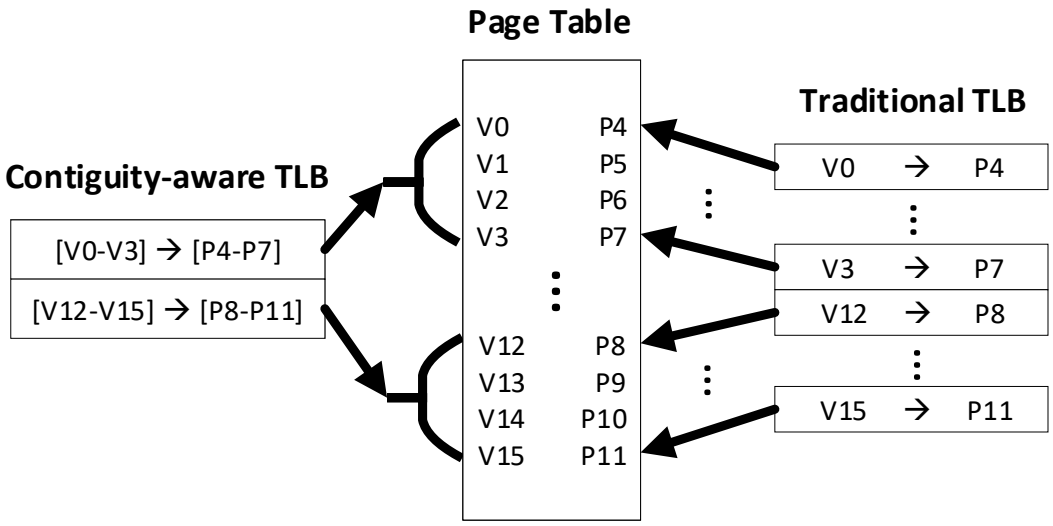


Figure 4.2: A contiguity-aware TLB (left) uses two entries to cache 4-page translation each. A traditional TLB (right) requires eight entries to cache the same number of translations.

customizations, like identity mappings, which can be difficult to create, preclude important OS features like copy-on-write, and may affect security features like address space layout randomization (ASLR) [59]. To achieve this, we propose Translation Ranger, an OS service that *actively coalesces fragmented pages* from the same virtually contiguous range to generate unrestricted amounts of physical memory contiguity in all scenarios outlined above. This enables previously proposed hardware TLB techniques—i.e., from COLT [30, 36, 125], direct segments [13], range TLB [76], hybrid coalescing TLB [121], to devirtualized memory [59]—to compress information about address translations into fewer hardware TLB entries. This permits us to realize area-efficient TLBs that scale gracefully with increasing memory capacity for accelerators (and CPUs). Our contributions are:

1. We are the first to propose active OS page coalescing to generate unbounded amounts of translation contiguity in all execution environments, in the presence or absence of memory fragmentation, or post-memory allocation. Because it does not depend on any special hardware, Translation Ranger's robust and general-purpose translation contiguity is widely applicable to all systems in real-world datacenter and cloud deployments with commercially-existing or recently-proposed

hardware techniques to compress entries in TLBs.

2. We implement Translation Ranger in Linux kernel v4.16 to assess the feasibility of our approach. Our real-system implementation and evaluation permits us to identify the subtle challenges of building Translation Ranger in commodity OSes. Chief amongst them are the challenges of reducing page migration overheads, dealing with the presence of pages deemed non-movable by the kernel, and understanding the impact of page coalescing on user application performance. We establish that it is useful to coalesce pages not only at allocation time but also post-allocation in highly-fragmented systems. This observation goes beyond all prior work which generally avoids post-allocation defragmentation because of its presumed overheads [13, 36, 59, 76, 125].

3. We show that Translation Ranger generates significant contiguity ($> 90\%$ of 120GB application footprint covered using only 128 contiguous regions, compared to $< 1\%$ without coalescing). It does so with very low overhead ($< 2\%$ of overall application runtime while coalescing 120GB memory), thereby ensuring that the performance gain delivered via coalescing is a net win.

4.2 Background

The increasing overheads of address translation and virtual memory have prompted significant academic and industry research on higher-performance address translation techniques [13, 30, 36, 59, 76, 121, 124, 125, 153]. Before we describe Translation Ranger, we first review this important background.

4.2.1 Specialized Contiguity-Aware Hardware

Several recent studies have proposed TLB hardware that can support arbitrary amounts of translation contiguity within each entry. These approaches generally task the programmer and OS with generating the necessary large swaths of contiguity. *Direct segments* [13] uses programmer-OS coordination to mark gigabyte- to terabyte-sized *primary segments* of memory which are guaranteed to maintain contiguous translation.

However, this approach presents challenges such as the need for one direct primary segment (real-world applications often require more flexibility in allocating contiguous segments in different parts of their address space) and explicit programmer intervention. *Redundant memory mappings* [76] support arbitrary translation ranges in TLBs; however, they do rely on aggressive OS changes to produce these translation ranges. *Devirtualized virtual memory* [59] extends the concepts of direct segments and vast translation contiguity for area-constrained accelerators, but it works under the somewhat optimistic assumption that OSes can always offer large contiguous memory regions for devirtualization.

The approaches above all require that large amounts of contiguity can be produced at allocation time, either via eager paging (which always allocates largest possible contiguous pages up to 2GB) or via boot-time memory reservation [13, 59, 76]. They also require non-standard changes to the OS kernel (e.g., custom memory allocators, use of position-independent executables, inefficient eager paging, etc.). Finally, these techniques are unable to react to dynamically-changing conditions or memory fragmentation, as discussed below. As such, these approaches are less general than Translation Ranger.

4.2.2 Improving Allocation-Time Contiguity

One approach to creating allocation-time contiguity is to simply reserve the memory in advance. With libhugetlbfs [91], memory is reserved at boot time and allocations for 2MB or 1GB pages are satisfied from these reserved memory pools. Device drivers often use customized memory allocators that perform similar advance reservation [35].

For allocations performed at runtime, the widely-used buddy memory allocator [80] is a source of translation contiguity because it groups contiguous free memory into free page pools of different sizes, from 1 to 2^N pages, where N is called the *max order* of the buddy allocator. Standard allocations will contain a maximum of 2^N contiguous pages. For example, Linux's buddy allocator supports maximum contiguous allocations of 4MB. However, because the buddy allocator must support fast insertions and deletions, free pages are stored in unordered lists, meaning that larger than 2^N pages worth of

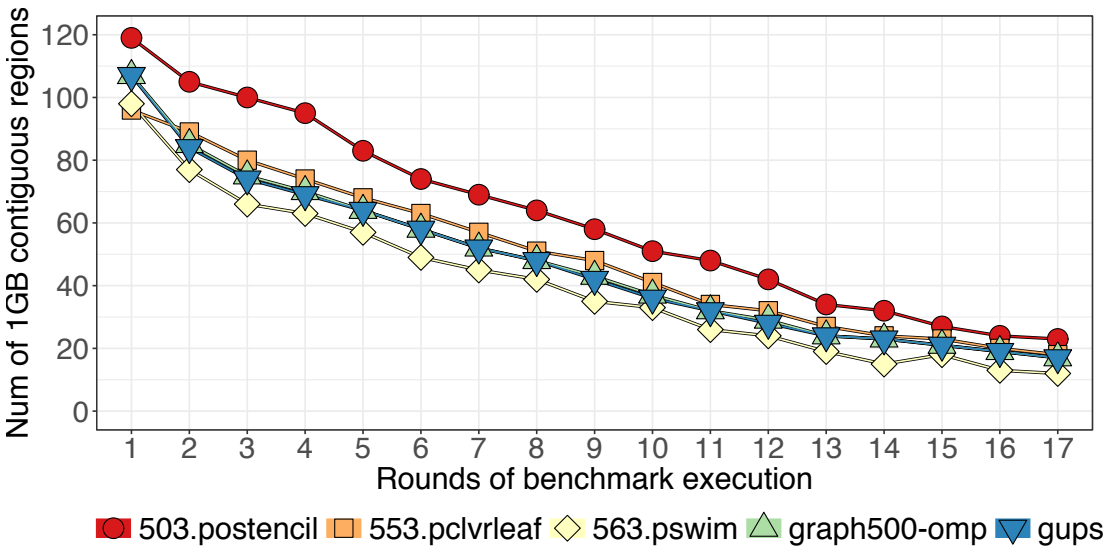


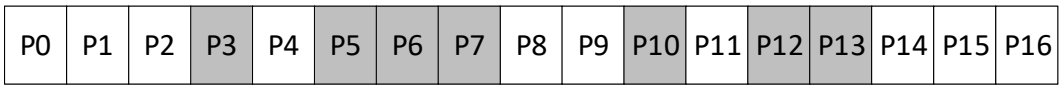
Figure 4.3: There is plenty of contiguity available at boot time, but memory becomes fragmented soon thereafter.

contiguity cannot be guaranteed upon sequential memory accesses even if two or more contiguous 2^N pages are in the same free list.

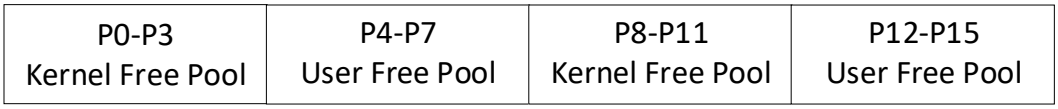
Previous work creates allocation-time contiguity by simply increasing the max order of the buddy allocator [59, 76]. However, this poses multiple problems. First, Linux's `sparsemem` (used to support discontiguous physical address spaces, which is common in modern systems), requires each contiguous physical address range to be aligned to 2^N [157]. This means, for example, that if the max order is increased to support contiguous free ranges of 1GB, gigabytes of available memory may be wasted. Second, increasing the max order does not solve the problem of fragmentation, i.e., the lack of contiguity of free pages. Fragmentation does not directly affect the contiguity of in-use memory ranges, but it can affect the amount of contiguity available at allocation time.

4.2.3 Memory Fragmentation and Defragmentation

To quantify the problem of fragmentation, we ran a set of benchmarks multiple times each, starting with a fresh-booted system. This system has 128GB memory (see Table 4.1), and we increased the max order of the buddy allocator to allocate large contiguous regions [59, 76]. Figure 4.3 shows that all benchmarks have majority (¿80%) of their footprint covered by 1GB contiguous regions (not to be confused with 1GB



(a) In-use page fragmentation prevents 8-page contiguity even though there are 10 free pages available. Filled blocks are in-use pages.



(b) Free page fragmentation prevents 8-page contiguity even though all pages are free.

Figure 4.4: Some possible types of fragmentation.

pages, which must also be aligned) on their first execution, right after the machine boots. However, as the benchmarks keep running, the number of 1GB contiguous regions decreases to only 20%. This degradation of large contiguous regions is often seen in long-run systems and is caused by memory fragmentation.

Fragmentation can occur for many reasons. In-use pages can prevent otherwise-free buddy pages from being promoted to a larger free page pool for allocation, as shown in Figure 4.4a. In-use pages allocated for use by the kernel are *non-movable* (also called *wired* in FreeBSD, *non-paged* in Windows [102,135]), which means that they cannot be defragmented [118]. The use of *kernel free page pools* can minimize the interleaving of kernel pages with user pages, as shown in Figure 4.4b; this avoids non-movable page fragmentation, but can cause free page fragmentation where kernel free page pools and user free page pools interleave with each other, preventing large contiguous regions from being formed [118]. This also explains why benchmarks in Figure 4.3 lose 1GB contiguous regions over multiple rounds of executions.

Perhaps surprisingly, defragmentation techniques can in practice turn out to have a detrimental effect on the contiguity of in-use memory. As shown in Figure 4.5, Linux uses memory compaction to move in-use pages to one end of physical address space, leaving the other end with contiguous free pages for high order free page promotions. However compaction moves only base pages and not transparent huge pages (THP), because traditional non-coalescing TLBs are unable to benefit from any higher contiguity. Therefore, it cannot assist in forming free page contiguity beyond 2MB size.

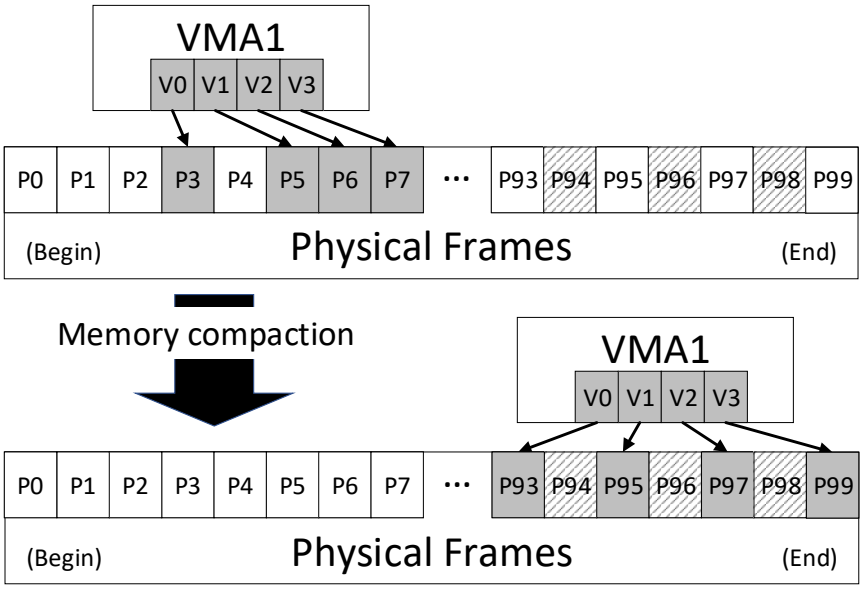


Figure 4.5: Defragmentation via memory compaction (e.g., in Linux) might destroy in-use contiguity as an unintended side effect of creating more free memory contiguity.

Compaction is also unaware of the contiguity of in-use pages, so if a set of contiguous in-use pages are moved to a set of scattered free pages, the original contiguity may be destroyed. Translation Ranger's goal is to show that instead, careful active coalescing can create contiguity post-allocation too, thereby overcoming the limitations of past approaches.

4.3 Translation Ranger

Previous sections established that in real-world settings, contiguity-based address translation optimizations will be highly sensitive to OS limitations such as memory fragmentation, lazy page allocation, and buddy allocator max order settings in the range of megabytes rather than gigabytes. Translation Ranger attacks these problems by gathering scattered physical pages that belong to a virtually contiguous range and rearranging them to achieve both virtual and physical contiguity. Unlike prior work, Translation Ranger generates contiguity at any granularity (including sizes between the architecturally-available page sizes), allowing it to scale with memory size with little runtime overhead.

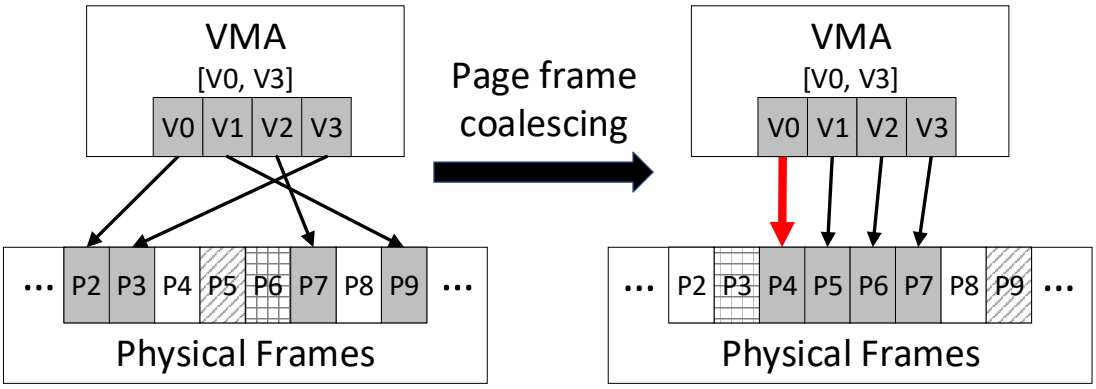


Figure 4.6: Coalescing the pages in a Virtual Memory Area (VMA): after coalescing page frames, virtual pages V0–V3 map to contiguous physical frames P4–P7. Filled page frame boxes denotes those mapped by V0–V3, marked boxes denotes the frames mapped by other VMAs, and blank boxes denotes free frames. The VMA's *Anchor Point* is (V0, P4).

4.3.1 Design Overview

We define a *contiguous region* as one where an arbitrary number of successive virtual pages map to an equal number of successive physical frames. The objective is that these contiguous translations can be leveraged with contiguity-aware TLBs (see Figure 4.2). Ideally, a contiguous region of N pages is cached with just a single TLB entry instead of N entries.

Contiguity in the virtual address space depends upon the layout of a process's virtual address space. Most OSes organize each process's virtual address space as multiple non-overlapping virtual address ranges. In Linux, each such virtual address range is described by the `struct vm_area_struct`, or virtual memory area (VMA). Applications obtain virtually-contiguous address ranges via `mmap` or `malloc`, but physical frames are allocated lazily upon the first access to each page. Contiguous regions are created when faulting virtual pages that are contiguous in a VMA are assigned contiguous physical frames. On-demand paging does not restrict the physical frame assigned to a faulting virtual page. Therefore, contiguous virtual pages in a VMA can be mapped to non-contiguous physical frames (see Figure 4.6).

Translation Ranger's approach is to rearrange the physical memory layout such that each VMA can be covered by as few contiguous regions as possible, with regions

that are as large as possible. Ideally, a single VMA would constitute one contiguous region, and would be tracked using just one TLB entry. To achieve this ideal situation, Translation Ranger follows three steps:

1. Actively coalesce memory within a VMA.
2. Avoid interference between the various VMAs in a process and in the overall system.
3. Iterate to maintain contiguity during the course of VMA allocations/expansions throughout application lifetime.

4.3.2 Intra-VMA Page Coalescing

As Figure 4.6 shows, Translation Ranger coalesces scattered physical frames into a contiguous region by first determining an *anchor point* for each VMA. An anchor point is a virtual page (VPN) and physical frame (PFN) pair, (V_{anchor}, P_{anchor}). After anchor point selection, Translation Ranger determines that to coalesce frames effectively, physical frames should be located at $P_n = (V_n - V_{anchor}) + P_{anchor}$ for each virtual page number V_n. Figure 4.6 shows an example where (V0, P4) is the anchor point. The VMA is coalesced by ① migrating P2 to P4, ② exchanging P5 with P9, ③ exchanging P7 with P6, ④ exchanging P7 with P3, ultimately leading to V0-V3 mapping to P4-P7. To do this correctly and efficiently, we must handle several key algorithmic issues:

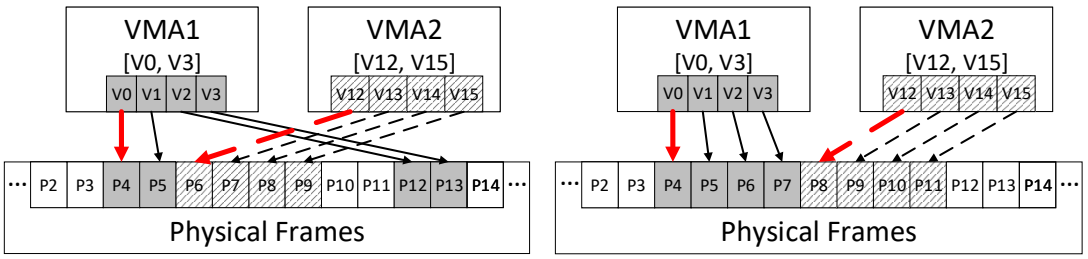
Targeting In-Use Page Frames. Target physical frames may be in one of two states: free or in-use. If a target page frame P_n is free, Linux's page migration mechanism is used to move the source frame to the target frame. If a target page frame P_n is in use, we cannot simply clobber it during coalescing. A simple solution would be to move the contents of the in-use frame itself to an intermediate physical frame before migrating the source frame to the target frame, but this suffers from extra storage and copy time overhead. In addition, when the system is under memory pressure, allocating a new intermediate physical frame can trigger the page reclamation process, leading to performance degradation. Therefore, we opt for an alternative approach:

we use an experimental patch available in Linux to directly exchange two anonymous pages [162]. This approach requires no extra storage and even supports the exchange of THPs. We further extended this patch to add support to exchange anonymous pages with file-backed pages, but we do not support exchange of two file-backed pages (due to complicated file system locking and their rare appearance).

Non-movable pages. Non-movable pages, which complicate page frame coalescing, can occur for several reasons. Some examples are that the page is used by the kernel (e.g., for SLAB allocations, page tables, or other kernel data structures), or that the page is busy (because it is involved in I/O operations, migration, dirty page write-back, or DMA). We use different approaches for kernel versus busy pages. Since kernel pages generally have long lifetimes and limited OS support for page migration, we skip them during the coalescing process. However, busy pages are generally only temporarily so; therefore, even if these pages cannot be moved on a first pass, Translation Ranger tries to coalesce them again in the future using its iterative process described in subsequent sections.

Upon skipping a non-movable page frame, Translation Ranger continues coalescing subsequent pages using the original anchor point. We investigated the value of creating new anchor points following non-movable pages, but we found that doing so yields little benefit. In fact, for busy pages, new anchor points can even be harmful if the page become non-busy in the near future. Creating the additional anchor point needlessly splits a large contiguous region into two smaller contiguous regions.

Anchor point alignment. When considering anchor point placement in a VMA, there is no constraint of alignment in VPN and PFN, which means any pair of VPN and PFN can be used as an anchor point. When Translation Ranger is used to improve contiguity in systems with traditional TLBs for 4KB, 2MB, and 1GB page sizes, to enable in-place promotion of 512 contiguous 4KB pages (or 2MB pages) to a single 2MB THP (or 1GB page), both the VPN and the PFN of the anchor point need to be aligned to 2MB (or 1GB) boundaries. In practice, we run experiments to align VMAs with sizes <1GB to 2MB boundaries and VMAs with sizes >1GB to 1GB



(a) Interference during coalescing (b) Avoiding interference

Figure 4.7: Anchor points must be chosen carefully in order to prevent inter-VMA interference.

boundaries and are able to obtain 2MB THPs out of aligned 512 4KB pages (1GB THPs are not supported in Linux). This means Translation Ranger is also effective on forming huge pages for systems with traditional TLBs. In terms of overall contiguity, we find that there is no difference in the effectiveness of page frame coalescing between using anchor points with any alignments and without alignments.

4.3.3 Avoiding Inter-VMA Interference

Achieving large contiguous regions also requires Translation Ranger to understand interaction between multiple VMAs. For example, consider Figure 4.7, where VMA1 contains V0-V3, while VMA2 contains V12-V15. If VMA1 and VMA2 use (V0, P4) and (V12, V8) as their respective anchor points, both VMAs can maximize the sizes of their contiguity regions. Figure 4.7a shows, however, that a less fortunate anchor point selection of (V12, P6) for VMA2 leads to poor contiguity formation. The problem is that these anchor points prompt both VMAs overlap in the physical frames that they must use to realize contiguous regions. In this situation, coalescing VMA2 would wipe out (some of) VMA1's contiguity and vice-versa. Furthermore, multiple coalescing passes would revisit this problem, creating excessive page migrations and increasing Translation Ranger's overheads.

To avoid this situation, Translation Ranger partitions the physical address space into disjoint coalesced and uncoalesced regions. Translation Ranger assigns each new VMA to an uncoalesced region. When this VMA is coalesced, the physical region to which it is assigned is marked coalesced. For this Translation Ranger

simply uses an algorithm similar to first-fit [159], i.e., scan linearly through all coalesced regions, and stop at the first sufficiently large hole between any two coalesced regions. This approach avoids inter-VMA interference, and thus substantially mitigates Translation Ranger overheads by minimizing unnecessary page migrations caused by physical range interference. This also provides useful information for tackling the problem of VMA size changes, which we subsequently discuss.

4.3.4 Iterative Page Frame Coalescing

Applications can grow and shrink their VMAs over time. This can be problematic when a VMA grows such that its physically contiguous region overlaps with another VMA, limiting contiguity. To avoid this problem, Translation Ranger tracks per-VMA size along with each VMA's coalesced region size during each coalescing iteration. If, on future iterations, Translation Ranger discovers that a VMA that has now grown and overlaps with another VMA (by examining coalesced region information), Translation Ranger relocates the coalesced region of one of the two VMAs by assigning a new anchor point. To minimize page frame relocation overheads, the smaller of the two is moved. On the other hand, if no overlapping occurs, the grown part of the VMA will be coalesced and the VMA's coalesced region size will be adjusted accordingly.

Second, Translation Ranger is designed to coalesce large important VMAs and ignore smaller shorter-lived VMAs (¡ 2MB) such as those used to map data structures like thread stacks. Our rationale was that these VMAs tend to be sufficiently small such that that coalescing their page frames them yields little additional contiguity relative to the overhead of the necessary page migrations. We believe more sophisticated strategies, e.g. consolidating multiple thread stacks, or lazy VMA deallocation to lengthen VMA lifetimes, could further decrease coalescing overhead and generate even larger contiguous regions, but we leave these for future work.

4.3.5 Multi-Process Coalescing and Synonyms

Finally, Translation Ranger coalesces page frames not for just one process, but all of them. For multi-process environments, Translation Ranger tracks the VMAs

Experimental Environment	
Processors	2-socket Intel E5–2650v4 (Broadwell), 24 cores/socket, 2 threads/core, 2.2 GHz
L1 DTLB	4KB pages: 64-entry, 4-way set assoc. 2MB pages: 32-entry, 4-way set assoc. 1GB pages: 4-entry, 4-way set assoc.
L1 ITLB	4KB pages: 128-entry, 4-way set assoc. 2MB pages: 8-entry, fully assoc.
L2 TLB	4KB&2MB pages: 1536-entry, 6-way set assoc. 1GB pages: 16-entry, 4-way set assoc.
Memory	128GB DDR4 (per socket)
OS	Debian Buster — Linux v4.16.0

Table 4.1: System configurations and per-core TLB hierarchy.

present in all running processes, just as `khugepaged` does in Linux. Translation Ranger is designed to account for forked and shared memories, or any situation where a contiguous physical region is shared by multiple VMAs. Translation Ranger only coalesces one physical range in these cases and skips the synonym VMAs. This can be done efficiently within Linux, for example, by checking `anon_vma` for anonymous VMAs and `address_space` for file-backed VMAs.

4.4 Experimental Methodology

Translation Ranger's strength is that it is widely deployable on systems with and without fragmentation and can leverage any previously-proposed TLB hardware that supports translation contiguity [13,59,76,121,125]. Naturally, Translation Ranger's performance benefits will vary depending on the target system, as well as the particular contiguity-aware TLB employed on that system. To achieve good performance, Translation Ranger must generate enough translation contiguity that contiguity-aware TLBs can leverage it to offset any runtime overheads imposed by Translation Ranger's page movement operations.

4.4.1 Evaluation Platform

We implement Translation Ranger in Linux kernel v4.16 and evaluate it on a two-socket Intel server (see Table 4.1). We run benchmarks from SPEC ACCEL [72], graph500, and GUPS in (see Table 4.2). Translation Ranger is tuned to periodically coalesce application memory; to produce contiguity statistics, application memory is scanned every 5 seconds to retrieve the virtual-to-physical mappings of each page belonging to the application. This statistics collection is engineered to have negligible impact on runtime.

One of Translation Ranger's goals is to generate contiguity on machines with realistic loads and uptimes, and with reasonable memory fragmentation. To eliminate the boot-time bonus we saw before executing our tests in Section 4.2.3, we induce fragmentation by compiling the Linux kernel multiple times; this allocates many file-backed pages, triggering SLAB allocations in the kernel. We also run a synthetic benchmark, memhog, an application that allocates memory all over the physical address space, and has been used in prior studies to create fragmentation [36, 124]. These preconditioning step ensures that memory in kernel and user free pools are in a fragmented state similar to a realistic steady state shown in Figure 4.3. To minimize artificial in-use page fragmentation caused by kernel compilations, we also release all file-backed pages and the corresponding kernel caches (e.g. `inodes` and `dentries`) by executing `sync` followed by `echo 3 > /proc/sys/vm/drop_caches` [90]. We configure our benchmarks to use around 95% of the total free memory. This usage is also typical of HPC systems where job schedulers attempt to match applications with available memory to prevent underutilization.

4.4.2 Experimental Configurations

To understand Translation Ranger's effectiveness on improving memory contiguity, we use Linux's default buddy allocator configuration (**Linux Default**) as our baseline, because most contiguity-aware TLB designs rely on serendipitously generated contiguity [121, 124, 125]. We also compare against an enhanced buddy allocator with its max order increased to 20, permitting 2GB contiguous region allocations. This **Large**

Suite	Description	Benchmark	Footprint
SPEC ACCEL	compute & memory intensive multi-threaded workloads	503.postencil	121GB
		551.ppalm	121GB
		553.pclvrleaf	121GB
		555.pseismic	120GB
		556.psp	113GB
		559.pmniGhost	120GB
		560.pilbdc	121GB
		563.pswim	117GB
		570.pbt	119GB
HPC	Generation and search of graphs	Graph500	122GB
	Random access benchmark	GUPS	128GB

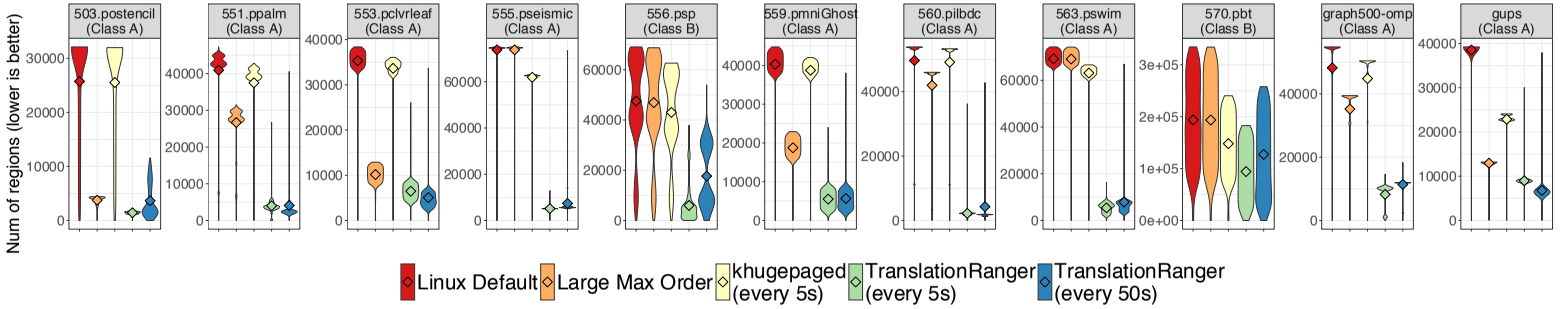
Table 4.2: Benchmark descriptions and memory footprints.

Max Order is representative of approaches from prior work like redundant memory mappings and devirtualized memory because it relies on generating contiguity at allocation time and hence serves as a valuable point of comparison to our approach [59, 76]. Furthermore, we compare Translation Ranger to **khugepaged**, which is another technique Linux uses to generate contiguity by collapsing scattered 4KB pages into a new THP. To conservatively assess Translation Ranger's relative benefits, we tune **khugepaged** to scan the entire application footprint every 5s as opposed to its default of defragmenting only 16MB of memory every 60s.

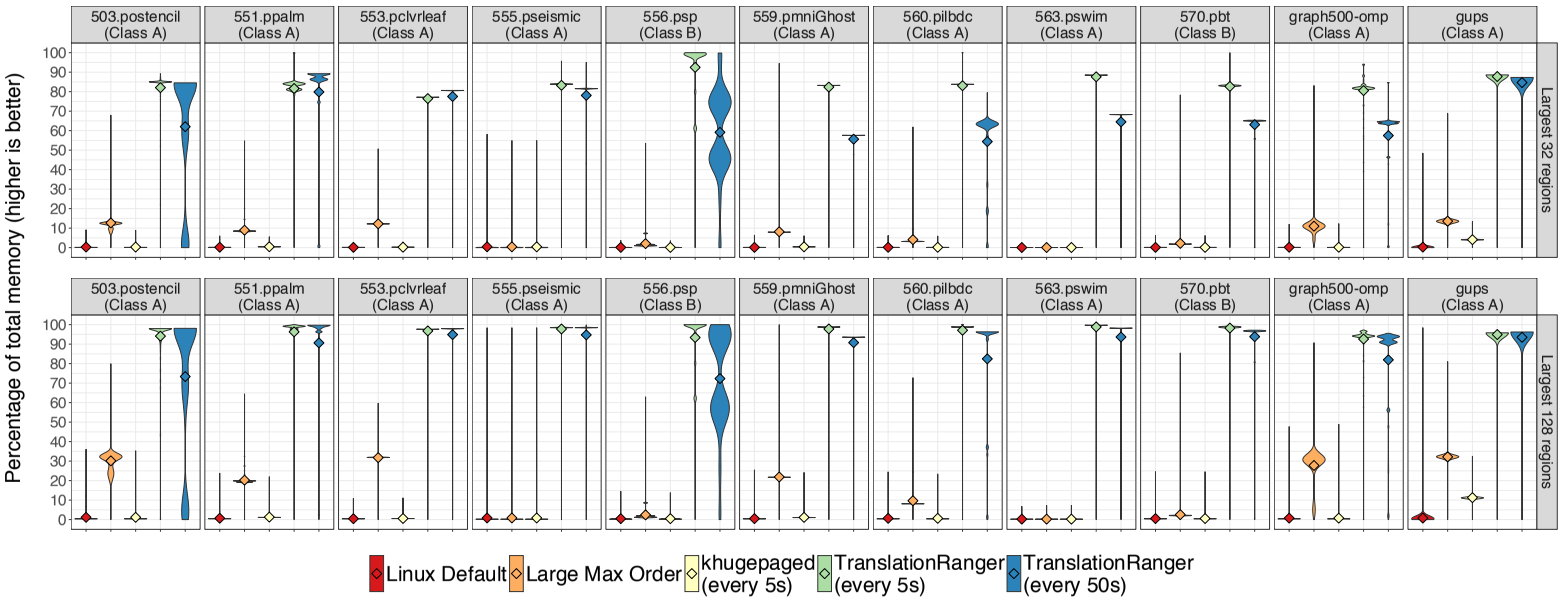
Finally, when profiling Translation Ranger, we quantify the results when coalescing every 5 seconds, and every 50 seconds to showcase the relationship between runtime overheads and Translation Ranger's ability to generate contiguity. For all five configurations, THPs are enabled by default and the buddy allocator uses Linux's default max order 11 to allow a maximum 4MB contiguous page allocation, except for **Large Max Order**.

4.4.3 Contiguity Metrics

We evaluate Translation Ranger using real-system experimentation, focusing on two key metrics. First, we count the total number of contiguous regions needed to cover the entire application memory footprint ($TotalNum_{ContigRegions}$). Our goal with



(a) Total number of contiguous regions covering entire application memory ($TotalNum_{ContigRegions}$).



(b) Percentage of total application footprint covered by the largest 32 contiguous regions ($MemCoverage_{32Regions}$) is shown in the top and percentage of total application footprint covered by the largest 128 contiguous regions ($MemCoverage_{128Regions}$) is shown in the bottom.

Figure 4.8: Contiguity results for all benchmarks.

this metric is to push the number of contiguous regions to quantities comparable to the most aggressive eager paging and identity mapping techniques from prior work (e.g., direct segments, range TLBs, devirtualization [13, 59, 76]). It is also to show that Translation Ranger can rival these techniques with respect to contiguity generation without sacrificing software and OS flexibility (by requiring the non-standard approaches the way these prior techniques do).

Next, we calculate the percentage of total application footprint covered by the largest 32 contiguous regions ($MemCoverage_{32Regions}$) and the percentage of total application footprint covered by the largest 128 contiguous regions ($MemCoverage_{128Regions}$). Our goal with this metric is to show that even small 32-128 entry contiguity-aware TLBs can capture the majority of the application footprint. This metric has been previously used by prior work [59, 76] to understand contiguity improvements.

4.4.4 Measuring Overhead

Translation Ranger's primary overhead comes from page migrations because pages undergoing migration are not accessible to applications, and necessitate frequent TLB invalidations and shootdowns. To understand these overheads, we present all benchmark runtimes for the five measured configurations and normalize them to our baseline, Linux Default. Our experimental platform supports discrete page sizes (4KB, 2MB, and 1GB) for address translation but does not take advantage of other kinds of contiguity, so excess application runtime due to coalescing can be viewed as the software tax of our system. Ultimately, the goal is to show that Translation Ranger generates contiguity comparable to the most aggressive of previous techniques, while simultaneously incurring such low overheads that the benefits are virtually "for free".

4.5 Experimental Results

We begin by showing translation contiguity results for all benchmarks. We follow that by highlighting two particularly interesting cases in order to provide more detail about how applications behave over time. Finally, we show the overhead.

Different memory allocation patterns within applications can have a large impact on contiguity generation. Based on profiling, we categorize our 11 benchmarks into two classes based on their memory allocation patterns:

Class A – Bulk memory allocation up front: 503.postencil, 551.ppalm, 553.pclvrleaf, 555.pseismic, 559.pmniGhost, 560.pilbdc, 563.pswim, graph500, and GUPS.

Class B – Continuous memory allocation over time: 556.psp and 570.pbt.

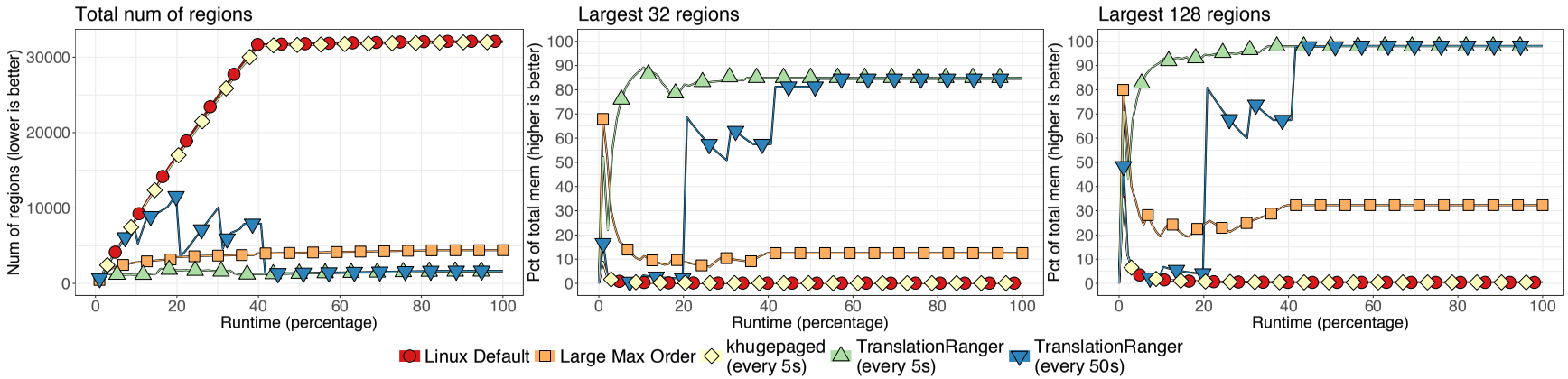
Class A is a very common HPC workload pattern where the majority of application memory is allocated at program inception before computation begins. Class B typically occurs when an application spawns many short-lived worker threads, each of which has its own private working memory pool.

4.5.1 Overall Translation Contiguity Results

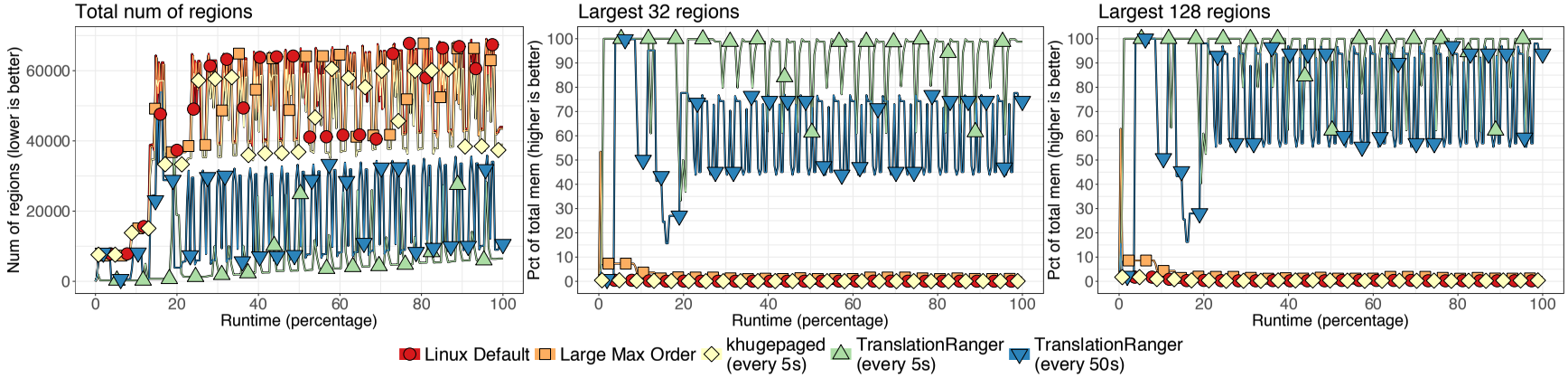
To concisely show individual results, we show $TotalNum_{ContigRegions}$, $MemCoverage_{32Regions}$, and $MemCoverage_{128Regions}$ in Figure 4.8 using Violin plots [158]. Violin plots aggregates all metric numbers over application runtime into a distribution "violin plot" parallel to the y-axis. The thickness of a violin plot indicates how often the value from y-axis occurs. In addition, the average number is shown as a diamond symbol within each plot.

Figure 4.8a shows the $TotalNum_{ContigRegions}$ distributions of all benchmarks. We observe that most Class A benchmarks show $TotalNum_{ContigRegions}$ aggregating in one primary area, which reflects their bulk memory allocation behavior; the number of regions does not change much over time. On the other hand, 556.psp and 570.pbt (Class B), which allocate memory recurrently, have their $TotalNum_{ContigRegions}$ spread across a range of values along the y-axis due to frequent small allocations.

Among all five configurations, Linux Default and khugepaged typically require many more contiguous regions (violin plots are thick at the top of each plot) to cover the application footprint, because they are both limited by the buddy allocator, which can only provide up to 4MB contiguous regions. We see that the Large Max Order configuration



(a) Contiguity results over time for 503.postencil (Class A).



(b) Contiguity results over time for 556.psp (Class B).

Figure 4.9: Total number of contiguous regions covering entire application memory ($TotalNum_{ContigRegions}$) is shown in the left most plot; percentage of total application footprint covered by the largest 32 contiguous regions ($MemCoverage_{32\,Regions}$) is shown in the middle plot; percentage of total application footprint covered by the largest 128 contiguous regions ($MemCoverage_{128\,Regions}$) is shown in the right most plot.

can generate much larger and thus fewer contiguous regions, since it modifies the buddy allocator to provide up to 2GB contiguous regions. However, Translation Ranger (at either frequency) is able to coalesce memory further and needs a far smaller number of contiguous regions to cover each application footprint; i.e., it is the *most successful* technique for generating contiguity.

Looking at Figure 4.8b for $MemCoverage_{32Regions}$ and $MemCoverage_{128Regions}$ distribution across all benchmarks, we see that unlike Figure 4.8a, the mean value of all benchmarks aggregates at certain point. This is because TLB coverage is influenced by both the size of the largest contiguous regions present in the system and not just the total number of regions. Among all five OS configurations, Linux Default and khugepaged typically only cover $< 1\%$ of each application footprint (with either 32 or 128 regions), since they can achieve at most 4MB contiguous regions. The Large Max Order configuration can typically cover up to 40% of the application footprint, but is also limited by 2GB contiguous regions from its buddy allocator modifications. Theoretically, the Large Max Order configuration should be able to cover all application footprints with 128 contiguous regions, if each region is at least 1GB. However, due to memory fragmentation, not all contiguous regions obtained from the buddy allocator are maximally-sized.

Translation Ranger (at frequencies of either 5 seconds or 50 seconds) consistently creates much larger contiguous regions that can cover the majority of each application's footprint. Utilizing 128 contiguous regions, Translation Ranger can typically cover $> 90\%$ of a 120GB application footprint. In comparison, the last level TLB of a CPU today typically contains 1024 entries and using entirely THPs (2MB) can only cover 2GB of footprint. The Translation Ranger approach combined with existing coalescing TLB proposals can improve TLB coverage by a factor of 20× while simultaneously decreasing TLB storage requirements by 10×.

4.5.2 Highlighting Individual Benchmarks of Interest

In order to provide more insight into how Translation Ranger achieves contiguity, we highlight one workload from each previously-defined class of applications: we pick

503.postencil to represent applications in Class A and 556.psp to represent Class B.

503.postencil (Class A)

Figure 4.9a shows the contiguity results over the runtime of 503.postencil, which first creates a huge address region, fills it with physical frames, then processes all data in memory. The left most plot in Figure 4.9a shows the $TotalNum_{ContigRegions}$ over application runtime for the 5 contiguity producing approaches. This plot shows the same data as the left most plot of Figure 4.8a, but in different forms. To translate between them, first consider the left most plot of Figure 4.9a. We can see $TotalNum_{ContigRegions}$ for Linux Default increases from 0 to about 32,000 during the first 40% of application runtime and becomes stable for the remaining 60% runtime. In the left most plot of Figure 4.8a, the Linux Default violin plot is thickest around the value of 32,000, then is fairly uniformly distributed between 0 and 32,000.

From the leftmost plot in Figure 4.9a, we see several interesting trends. For example, with Translation Ranger more frequent invocation (every 5 seconds) is able to generate large contiguous regions more quickly than the less frequent invocation (every 50 seconds). However, after application memory allocations become stable (at approximately 40% of the application runtime), the less frequent invocation eventually results in a similar number of contiguous regions being produced for the remainder of the execution.

In terms of contiguous region TLB coverage, the middle and right most plots of Figure 4.9a show $MemCoverage_{32Regions}$ and $MemCoverage_{128Regions}$ over application runtime, respectively. We observe that the Linux Default and khugepaged configurations can cover very little ($< 1\%$) of the application footprint with even 128 contiguous regions. The Large Max Order configuration is a substantial improvement with 12.5% of the application footprint being covered with the largest 32 contiguous regions and 32.3% with the largest 128 contiguous regions respectively. However Translation Ranger can cover at least 80% of application footprint with just 32 contiguous regions, and over 95% of the 121GB footprint using 128 contiguous regions.

To summarize, for 503.postencil, contiguity-aware TLB designs will attain 3–6×

more TLB coverage (or could reduce their hardware resources proportionally), if they simply use Translation Ranger instead of their custom software solutions.

556.psp (Class B)

Figure 4.9b shows the contiguity results for 556.psp, which frequently allocates and deallocates memory from within spawned worker threads. The left most plot of Figure 4.9b shows high volatility across all configurations for $TotalNum_{ContigRegions}$ because frequent memory allocations add many fragmented small pages and deallocations remove existing contiguous regions. All five experimental configurations suffer notably from this type of application memory allocation pattern.

Because Translation Ranger iteratively coalesces memory to generate contiguous regions, it requires 40% fewer contiguous regions to cover the entire application footprint than Linux Default, Large Max Order, or khugepaged. The middle and right plots in Figure 4.9b show that using the largest 32 or 128 contiguous regions, Linux Default and khugepaged can cover $< 1\%$ of the application footprint. Large Max Order does a little better, covering about 8% initially, but decreases to $< 2\%$ because of the frequent memory allocations and deallocations.

In contrast, Translation Ranger (running every 5 seconds) covers more than 80% of the application footprint and up to 99% during the bulk of application runtime when considering 32 regions or 128 regions. Translation Ranger (with frequency set to every 50 seconds) can cover more than 45% of the application footprint with the largest 32 contiguous regions and the coverage is over 75% during the bulk of application runtime; increasing the number of regions to 128 further improves the coverage of the application's footprint to 94%. It is clear that Translation Ranger provides a significant advantage over the alternatives when trying to generate contiguity.

4.5.3 Low Translation Ranger Overheads

So far we have seen that Translation Ranger creates systematically larger and dramatically fewer contiguous regions compared to other approaches, but we must also

consider the runtime overhead it imposes on applications. Figure 4.10 shows the application execution time across all five configurations normalized to our baseline, Linux Default, when run 5 times to account for runtime variation. Large Max Order, which on average adds 0.4% runtime overhead, incurs very minor slowdowns due to zeroing every free large page at allocation time. The overhead of khugepaged is negligible. Although it runs very frequently (every 5 seconds), the majority of each workload memory are already THPs, leaving few base pages for it to convert to THPs. This also explains its small improvements in contiguity for most workloads.

We find that Translation Ranger adds, on average, only 1.9% overhead when it runs every 5 seconds and 1.2% overhead when it runs every 50 seconds. In the worst case, 556.psp suffers from 6.7% overhead when Translation Ranger runs every 5 seconds. However, not only can this overhead be lowered to 0.6% once we decrease Translation Rangerś frequency to every 50 seconds, we can do this while effectively enjoying similar contiguity creation as the 5 second case. Graph500 also suffers a 5.1% overhead when Translation Ranger runs too frequently, but the overhead is reduced to 1.7% by reducing Translation Ranger frequency (again, without noticeable drop in contiguity). Though Translation Ranger has slightly higher software overheads than previously proposed solutions (represented by Large Max Order), it is typically under 1%, which will be more than offset by the performance improvements achieved via TLB efficiency, reported at 20-30% in prior proposals [59, 76, 125]. Because Translation Ranger generates substantially more contiguity than used in these studies, we would expect to see more performance but an exhaustive study of contiguity-aware TLB designs is beyond the scope of this work.

4.5.4 Summary

From the translation contiguity results and runtime results, we conclude Translation Ranger is a low overhead and highly effective technique to generate translation contiguity in systems with fragmented memory, particularly when compared to three other approaches on which prior contiguity-aware TLB designs rely. Translation

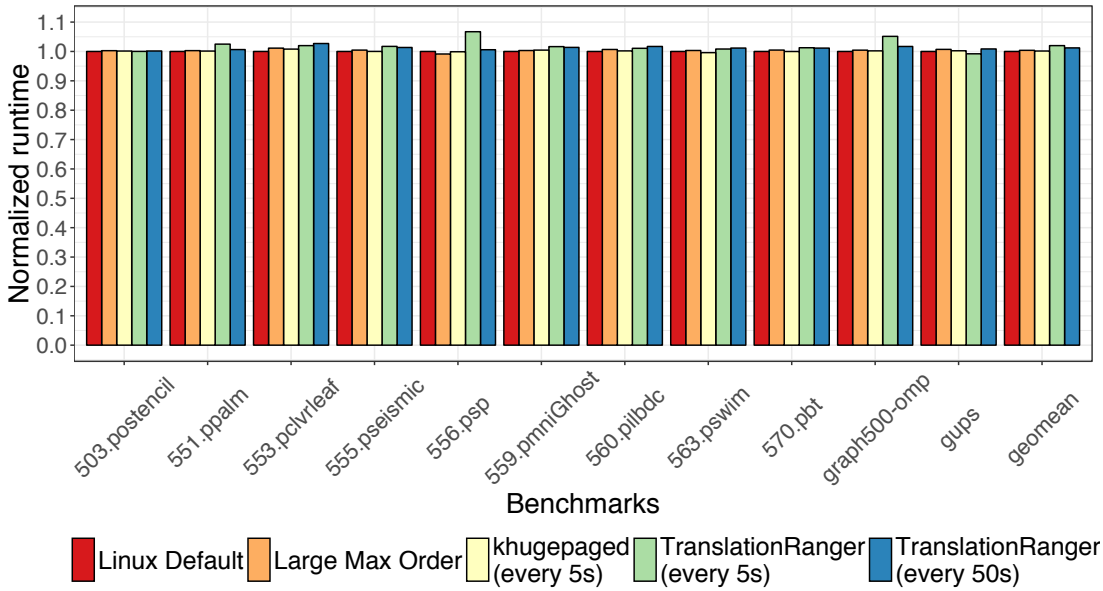


Figure 4.10: Benchmark runtime for all five configurations: Linux Default, Large Max Order, khugepaged, and Translation Ranger with two running frequency. All runtime is normalized to Linux Default.

RANGERachieves the following —it covers $>$ 60% application footprint with 32 contiguous regions and $>$ 90% with 128 contiguous regions.

4.6 Discussion

Applicability to other OSes. TRANSLATION RANGER is compatible with other OSes like Windows and FreeBSD, which maintain VMA-like data structures per process. For example, FreeBSD uses `vm_map_entry` to identify a contiguous virtual address range instead of a VMA, `vm_object` to represent a group of physical frames from one memory object instead of `anon_vma` for an anonymous memory object and `address_mapping` for a file in Linux. To port TRANSLATION RANGER to FreeBSD, we can generate contiguity on each `vm_map_entry` and assign one anchor point to each `vm_map_entry` instead of each VMA. Extra care needs to be taken when generating contiguity on pages created by copy-on-write (COW). FreeBSD uses *shadow* `vm_object`s to keep track of private pages created at COW, thus we should skip the pages in *shadow* `vm_object`s during memory defragmentation to avoid unnecessary work, the same way TRANSLATION RANGER does for skipping similar VMAs.

Permission checks. Contiguity generated by Translation Ranger and other techniques belongs to one VMA, which has a uniform permission across all virtual addresses in it. When the permission of a virtual address range in this VMA is changed, the VMA will be broken into two or more new VMAs with corresponding updated permissions, and the page table entries will be updated respectively. This also breaks one contiguous region into multiples, and contiguity-aware TLBs will need additional entries to address the original memory address range. To mitigate this problem, new address translation designs [1, 13] can decouple permission checks from virtual-to-physical address translation, could be helpful, allowing the TLB to maintain the original contiguous region entry but with additional permission subsections.

NUMA effects. Our initial study focuses on single-socket systems as a tractable configuration to thoroughly understand Translation Ranger. However, Translation Ranger could be extended to NUMA systems. In fact, Translation Ranger's cross-socket traffic overheads can likely be tamed by previously-proposed NUMA paging policies, like autoNUMA, Carrefour-LP, and Ingens [34, 50, 81].

4.7 Related Work

There exists a wealth of work on memory allocation to increase contiguity and reduce memory fragmentation [25, 70, 104]. Table 4.3 summarizes these contiguity-based techniques and discusses the requirements they impose on system software, as well the hardware support they require and their ability to cover memory capacity.

Memory allocation. Internal and external memory fragmentations are two major problems for memory allocations. To mitigate internal memory fragmentation, SLAB allocator and others (e.g., jemelloc and tcmellaoc) pack small memory objects with the same size together in one or more pages to avoid wasting space internally [25, 43, 51]. For external memory fragmentation, OSes use buddy allocators to achieve fast memory allocation and restrict external fragmentation [80]. Linux developers also have other heuristics to further reduce external fragmentation [54]. Additionally, peripheral devices often require access to physically contiguous memory. Linux accommodates these

Techniques	Software Requirements			Hardware Requirements	Coverage Limits
	Buddy Allocator	Reservation	Page Table		
hugetlbfs	Separate pools	Required	No change	2MB & 1GB page size TLB	2MB and 1GB
THP & khugepaged	2MB pages	None	No change	2MB page size TLB	2MB
COLT [125]	Contiguous pages	None	No change	Coalesced TLB	Up to 8× 4KB
Dir Segments [13]	Not related	Required	Segment table	Direct Segment registers	Any size
Redundant Mem Mappings [76]	Increase max order, eager paging	None	Range table	Range TLB	Any size in HW, but limited by SW
Hybrid TLB [121]	Contiguous pages	None	Anchor page table	Hybrid TLB	$N\times$ 4KB or 2MB
Devirtualized VM [59]	Increase max order, eager paging	None	A new page table entry	Access validation cache	4KB, 2MB, or 1GB in HW, but limited by SW

Table 4.3: Techniques used/proposed by industrial or academic research groups for high performance address translation.

devices with drivers that use boot-time allocation and reserve contiguous memory before others can request memory via Contiguous Memory Allocators (CMAs) [35]. CMAs use memory compaction to migrate fragmented pages and offer large contiguous physical memory for devices use, especially for DMA data transfer [104]. These approaches try to preserve contiguity for future use, but cannot prevent large free memory blocks from being broken into small ones when memory requests are small in sizes.

Huge pages. Significant prior work has gone into improving the performance of huge pages. For example, huge page has only one access bit, causing memory access imbalance implementation problems in NUMA systems [50]. Several proposals try to manage huge page wisely by restricting huge page creation and splitting huge pages when they cause utilization imbalance issues [50, 81]. These utilization issues could also happen on contiguous regions created by Translation Ranger, thus integrating these policies with Translation Ranger could further improve application performance in NUMA systems.

In heterogeneous memory systems, classifying hot and cold pages to determine how

to move them between fast/slow memories is important. Previous work by Thermostat analyzes huge page utilization by sampling sub-pages in each huge page and migrates these cold sub-pages to slow memory to make efficient use of fast memory [4]. Thermostat could provide useful utilization information to Translation Ranger to assess page hotness in identifying which VMAs are particularly worth defragmenting.

The high cost of huge page allocation has been a long standing problem for Linux and can lead to application performance degradation [42, 99]. Recent work studies methods in which Linux can prevent free page fragmentation and eliminate most huge page allocation costs by aggregating kernel page allocations [118]. With the help of this work, Translation Ranger could also improve in-use page fragmentation and free page fragmentation, thereby generating even larger contiguous regions.

4.8 Conclusions

Translation Ranger is a low overhead and effective technique of coalescing scattered physical frames and generating translation contiguity. The enormous contiguous regions created by Translation Ranger can be used by emerging contiguity-aware TLBs to dramatically minimize address translation overhead for all computation units in heterogeneous systems, especially accelerators, which have limited hardware resources for address translation. Translation Ranger can scale easily with increasing memory sizes regardless of the limitations imposed by memory allocators. With less than 2% runtime overhead, Translation Ranger generates contiguous regions covering more than 90% of 120GB application footprints with at most 128 regions, which can be fully cached by contiguity-aware TLBs to minimize high address translation overhead. To address ever-increasing memory sizes, contiguity-aware TLBs provide promising hardware support and Translation Ranger is the software cornerstone needed to support contiguity-aware TLBs.

Chapter 5

Conclusions

Virtual memory, as powerful hardware abstraction, has existed for decades and keeps evolving. With the emergence of heterogeneous memory systems, virtual memory needs an upgrade for the new challenges posed by the systems. This dissertation has addressed one of the most critical challenges, how to upgrade virtual memory support for heterogeneous memory management. The thesis of this dissertation is that page migration is an essential operation for heterogeneous memory management, hence accelerating page migration and making the intelligent use of it is the key to attain high performance in heterogeneous memory systems. To support the thesis, several contributions that reduce the overheads of virtual memory brought by the introduction of heterogeneous memory systems are made in this dissertation, i.e. eliminating the address translation coherence overheads from frequent page migration, accelerating operating system page migration process for high throughput data movement, and utilizing fast page migration to coalesce memory for scalable address translation coverage.

Although this dissertation focuses on two different parts, address translation coherence and page copy process, of page migration and the intelligent use of it, memory coalescing to increase address translation coverage individually, the optimizations on both parts of page migration can be easily combined to achieve even higher performance in page migration, while the memory coalescing is also able to lower the virtual memory overhead during page migration in turn. A good fusion of all three techniques proposed by this dissertation is the key to managing heterogeneous memories. It requires careful tunings of these three techniques depending on the cache coherence protocol used in the system, the availability and capability of data copy hardware, like a DMA data copy engine or a process with high-throughput data movement instructions, and the ways

that the system TLBs are utilizing address translation contiguity. Different methods of maintaining cache coherence impacts address translation coherence when hardware translation coherence is used, which is also influenced by the means of translation structures, like TLBs, compressing translation information, because the scope of translation coherence is widened by condensed translation information. In a system with DMA data copy engines, the data copy process in page migration no longer consumes CPU resource, which changes page migration policies in a way that they do not need to consider the CPU resource competition between page migration and applications. These are the tradeoffs, which give significant performance impacts in heterogeneous memory systems.

Although this dissertation targets heterogeneous memory systems, all techniques proposed may apply to conventional homogeneous memory systems, like NUMA systems. For example, memory fragmentation has been a long-standing problem for virtual memory, which can be alleviated by the memory coalescing technique from Chapter 4. Using page migration to decrease memory fragmentation is not uncommon, Linux has used page migration to compact memory for less fragmentation [33].

In summary, the research from this dissertation shows that high-performance heterogeneous memory systems can be attained as virtual memory keeps adapting to the new system demands via fast page migration and the intelligent use of it.

References

[1] Reto Achermann, Chris Dalton, Paolo Faraboschi, Moritz Hoffmann, Dejan Milojicic, Geoffrey Ndu, Alexander Richardson, Timothy Roscoe, Adrian L. Shaw, and Robert N. M. Watson. Separating translation from protection in address spaces with dynamic remapping. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 118–124, New York, NY, USA, 2017. ACM.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 2–13, New York, NY, USA, 2006. ACM.

[3] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 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, pages 607–618, New York, NY, USA, 2015. ACM.

[4] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 631–644, New York, NY, USA, 2017. ACM.

[5] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting hardware-assisted page walks for virtualized systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 476–487, Washington, DC, USA, 2012. IEEE Computer Society.

[6] AMD Corporation. Compute Cores. `https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf`, 2014. [Online; accessed 04-Aug-2018].

[7] Andrea Arcangeli. Rfc: Transparent hugepage support. `https://lwn.net/Articles/358904/`. [Online; accessed 31-Jul-2018].

[8] Andrea Arcangeli. Transparent Hugepage Support. *KVM Forum*, August 2010.

[9] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 416–427, Washington, DC, USA, 2012. IEEE Computer Society.

[10] Amitabha Banerjee, Rishi Mehta, and Zach Shen. Numa aware i/o in virtualized systems. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, HOTI '15, pages 10–17, Washington, DC, USA, 2015. IEEE Computer Society.

[11] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 48–59, New York, NY, USA, 2010. ACM.

[12] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Spectlb: A mechanism for speculative address translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 307–318, New York, NY, USA, 2011. ACM.

[13] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[14] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society.

[15] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 26–35, New York, NY, USA, 2008. ACM.

[16] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 383–394, New York, NY, USA, 2013. ACM.

[17] Abhishek Bhattacharjee. Preserving virtual memory by mitigating the address translation wall. *IEEE Micro*, 37(5):6–10, September 2017.

[18] Abhishek Bhattacharjee. Translation-triggered prefetching. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 63–76, New York, NY, USA, 2017. ACM.

[19] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level tlbs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 62–63, Feb 2011.

[20] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[21] 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. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 469–479, Washington, DC, USA, 2006. IEEE Computer Society.

[22] David L Black and Daniel D Sleator. Competitive algorithms for replication and migration problems. Carnegie-Mellon University Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[23] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. Concurrent page migration for mobile systems with os-managed hybrid memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 31:1–31:10, New York, NY, USA, 2014. ACM.

[24] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 19–31, New York, NY, USA, 1989. ACM.

[25] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[26] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 12–24, New York, NY, USA, 1994. ACM.

[27] Kevin Chang, Donghyuk Lee, Zeshan Chishti, Alaa Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving dram performance by parallelizing refreshes with accesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 356–367, Feb 2014.

[28] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, pages 268–280, New York, NY, USA, 2017. ACM.

[29] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 1–12, Washington, DC, USA, 2014. IEEE Computer Society.

[30] Mike Clark. A new x86 core architecture for the next generation of computing. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–19, Aug 2016.

[31] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Evaluation of the memory page migration influence in the system performance: The case of the sgi o2000. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 121–129, New York, NY, USA, 2003. ACM.

[32] Jonathan Corbet. Heterogeneous memory management. `http://lwn.net/Articles/684916`.

[33] Jonathan Corbet. Memory compaction. `https://lwn.net/Articles/368869/`.

[34] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. `http://lwn.net/Articles/488709/`, 2012. [Online; accessed 31-Jul-2018].

[35] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[36] Guilherme Cox and Abhishek Bhattacharjee. 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, pages 435–448, New York, NY, USA, 2017. ACM.

[37] Kathy Davies. What's new in hyper-v on windows server 2016 technical preview. `https://technet.microsoft.com/en-us/windows-server-docs/compute/hyper-v/what-s-new-in-hyper-v-on-windows`, August 2016. [Online; accessed: 31-Jul-2018].

[38] Peter J. Denning. The working set model for program behavior. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 15.1–15.12, New York, NY, USA, 1967. ACM.

[39] Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. A circuit-architecture co-optimization framework for exploring nonvolatile memory hierarchies. *ACM Trans. Archit. Code Optim.*, 10(4):23:1–23:22, December 2013.

[40] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[41] M. C. Easton and P. A. Franaszek. Use bit scanning in replacement decisions. *IEEE Transactions on Computers*, C-28(2):133–141, Feb 1979.

[42] Nelson Elhage. Disable transparent hugepages. `https://blog.nelhage.com/post/transparent-hugepages/`. [Online; accessed 04-Aug-2018].

[43] Jason Evans. Scalable memory allocation using jemalloc. `https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919`, 2011. [Online; accessed 04-Aug-2018].

[44] Babak Falsafi, Tim Harris, Dushyanth Narayanan, and David A. Patterson. Rack-scale Computing (Dagstuhl Seminar 15421). *Dagstuhl Reports*, 5(10):35–49, 2016.

[45] Dongrui Fan, Zhimin Tang, Hailin Huang, and Guang R. Gao. An energy efficient tlb design methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 351–356, New York, NY, USA, 2005. ACM.

[46] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 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, pages 37–48, New York, NY, USA, 2012. ACM.

[47] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News*, 42(2):20–23, September 2014.

[48] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 178–189, Washington, DC, USA, 2014. IEEE Computer Society.

[49] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 707–718, Piscataway, NJ, USA, 2016. IEEE Press.

[50] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[51] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2009. [Online; accessed 04-Aug-2018].

[52] Jerome Glisse. HMM (Heterogeneous memory management) v5. `http://lwn.net/Articles/619067`.

[53] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Bruce Perens' Open source series. Prentice Hall, 2004.

[54] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*, pages 369–384, 2006.

[55] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, 2017. USENIX Association.

[56] Nagendra Gulur, Mahesh Mehendale, R. Manikantan, and R. Govindarajan. Bimodal dram cache: A scalable and effective die-stacked dram cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 38–50, Washington, DC, USA, 2014. IEEE Computer Society.

[57] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 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, pages 39–51, New York, NY, USA, 2015. ACM.

[58] Vishal Gupta, Min Lee, and Karsten Schwan. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 79–92, New York, NY, USA, 2015. ACM.

[59] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing memory in heterogeneous systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 637–650, New York, NY, USA, 2018. ACM.

[60] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[61] HSA Foundation. HSA Platform System Architecture Specification - Provisional 1.0. `http://www.slideshare.net/hsafoundation/hsa-platform-system-architecture-specification-provisional-verl-10-ratifed`, 2014. [Online; accessed 04-Aug-2018].

[62] Micron Technology Inc. 3D XPoint Technology. `https://www.micron.com/products/advanced-solutions/3d-xpoint-technology`, 2016.

[63] Intel. Intel memory latency checker. `https://software.intel.com/en-us/articles/intelr-memory-latency-checker`. [Online; accessed 31-Jul-2018].

[64] Intel. Introducing Intel Optane Technology - Bringing 3D XPoint Memory to Storage and Memory Products, 2015.

[65] Intel. Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor. `http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf`, 2016. [Online; accessed 31-Jul-2018].

[66] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. April 2018.

[67] JEDEC. JESD79-4A: DDR4 SDRAM Standard. `https://www.jedec.org/sites/default/files/docs/JESD79-4A.pdf`, November 2014. [Online; accessed 31-Jul-2018].

[68] JEDEC. High Bandwidth Memory(HBM) DRAM - JESD235A. `http://www.jedec.org/standards-documents/docs/jesd235a`, November 2015. [Online; accessed 31-Jul-2018].

[69] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 25–37, Washington, DC, USA, 2014. IEEE Computer Society.

[70] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 26–36, New York, NY, USA, 1998. ACM.

[71] Toni Juan, Tomas Lang, and Juan J. Navarro. Reducing tlb power requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 196–201, New York, NY, USA, 1997. ACM.

[72] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. *SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance*, pages 46–67. Springer International Publishing, Cham, 2015.

[73] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating physical addresses directly for saving instruction tlb energy. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 185–196, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[74] Ajaykumar Kannan, Natalie Enright Jerger, and Gabriel H. Loh. Enabling interposer-based disintegration of multi-core processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 546–558, New York, NY, USA, 2015. ACM.

[75] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. ACM.

[76] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 66–78, New York, NY, USA, 2015. ACM.

[77] Vasileios Karakostas, Jayneel Gandhi, Adrian Cristal, Mark Hill, Kathryn McKinley, Mario Nemirovsky, Michael Swift, and Osman Unsal. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 631–643, March 2016.

[78] Anshuman Khandaul. Define coherent device memory node. `http://lwn.net/Articles/404403`, 2016.

[79] Joonyoung Kim and Younsu Kim. Hbm: Memory solution for bandwidth-hungry processors. *2014 IEEE Hot Chips 26 Symposium (HCS)*, 00:1–24, 2014.

[80] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[81] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 705–721, Berkeley, CA, USA, 2016. USENIX Association.

[82] Christoph Lameter. Swap migration v3: Overview. `https://lwn.net/Articles/156603/`. [Online; accessed 31-Jul-2018].

[83] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[84] Lawerence Livermore National Laboratory. CORAL/Sierra. `https://asc.llnl.gov/coral-info`, 2016. [Online; accessed 31-Jul-2018].

[85] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[86] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 277–289, Berkeley, CA, USA, 2015. USENIX Association.

[87] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. ACM.

[88] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level Implications of Disaggregated Memory. In *Intl. Symp. on High Performance Computer Architecture*, pages 1–12, February 2012.

[89] Felix Xiaozhu Lin and Xu Liu. Memif: Towards programming heterogeneous memory asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 369–383. ACM, 2016.

[90] Linux. Documentation for /proc/sys/vm/*. `https://www.kernel.org/doc/Documentation/sysctl/vm.txt`. [Online; accessed 04-Aug-2018].

[91] Adam Litke. libhugetlbfs. `https://lwn.net/Articles/171451/`. [Online; accessed 04-Aug-2018].

[92] Gabriel Loh and Mark D. Hill. Supporting very large dram caches with compound-access scheduling and missmap. *IEEE Micro*, 32(3):70–78, May 2012.

[93] Gabriel H. Loh and Mark D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 454–464, New York, NY, USA, 2011. ACM.

[94] Jason Lowe-Power. Inferring kaveri's shared virtual memory implementation. `http://www.lowepower.com/jason/inferring-kaveris-shared-virtual-memory-implementation.html`. [Online; accessed 04-Aug-2018].

[95] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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, pages 190–200, New York, NY, USA, 2005. ACM.

[96] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs. *ACM Trans. Archit. Code Optim.*, 10(1):2:1–2:38, April 2013.

[97] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 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, pages 233–247, New York, NY, USA, 2016. ACM.

[98] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting nvm in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 2. ACM, 2015.

[99] MongoDB Manual. Disable transparent hugepages (thp). `https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/`. [Online; accessed 04-Aug-2018].

[100] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[101] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[102] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[103] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, J ohn Slice, Mike Ignatowski, and Gabriel H. Loh. 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)*, pages 126–136, February 2015.

[104] Michal Nazarewicz. The Contiguous Memory Allocator. `https://lwn.net/Articles/396657/`, 2010. [Online; accessed 04-Aug-2018].

[105] Hybrid Memory Cube Specification 2.1. `https://www.nuvation.com/sites/default/files/Nuvation-Engineering-Images/Articles/FPGAs-and-HMC/HMC-30G-VSR_HMCC_Specification.pdf`, 2015. [Online; accessed 31-Jul-2018].

[106] Jeffery Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 14–18, Berkeley, CA, USA, May 2009. USENIX Association.

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

[108] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. In *Cray User's Group*, May 2010.

[109] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, December 2002.

[110] Kernel Newbie. Linux 4.14. `https://kernelnewbies.org/Linux_4.14#Memory_management`.

[111] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. A case for user-level dynamic page migration. In *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, pages 119–130, New York, NY, USA, 2000. ACM.

[112] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, ICPP '00, pages 95–, Washington, DC, USA, 2000. IEEE Computer Society.

[113] NVIDIA Corporation. Unified Memory in CUDA 6. `http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/`, 2013. [Online; accessed 31-Jul-2018].

[114] NVIDIA Corporation. NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data. `http://devblogs.nvidia.com/parallelforall/nvlink-pascal-stacked-memory-feeding-appetite-big-data/`, 2014. [Online; accessed 14-Aug-2016].

[115] Oak Ridge National Laboratory. Summit. `https://www.olcf.ornl.gov/summit/`, 2018. [Online; accessed 31-Jul-2018].

[116] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 188–200, Washington, DC, USA, 2015. IEEE Computer Society.

[117] Jiannan Ouyang, John R. Lange, and Haoqiang Zheng. 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, pages 17–23, New York, NY, USA, 2016. ACM.

[118] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 679–692, New York, NY, USA, 2018. ACM.

[119] Myrto Papadopoulou, Xin Tong, Andrew Seznec, and Andreas Moshovos. Prediction-Based Superpage-Friendly TLB Designs. *HPCA*, 2014.

[120] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. Seesaw: Using superpages to improve vipt caches. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 193–206, June 2018.

[121] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 444–456, New York, NY, USA, 2017. ACM.

[122] J. Thomas Pawlowski. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24, Aug 2011.

[123] Sujay Phadke and Satish Narayanasamy. MLP aware heterogeneous memory system. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[124] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 558–567, Feb 2014.

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

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

[127] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 1–12, New York, NY, USA, 2015. ACM.

[128] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 743–758, March 2014.

[129] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual International Symposium on Microarchitecture*, 2012.

[130] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[131] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[132] Dulloor Subramanya Rao and Karsten Schwan. vNUMA-mgr: Managing VM memory on NUMA platforms. In *2010 International Conference on High Performance Computing*, pages 1–10, Dec 2010.

[133] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. 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, pages 306–317, Washington, DC, USA, 2013. IEEE Computer Society.

[134] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. 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*, pages 1–12, Jan 2010.

[135] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 2: Covering Windows Server 2008 R2 and Windows 7 (Windows Internals)*. Microsoft Press, Redmond, WA, USA, 2012.

[136] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. A case for granularity aware page migration. In *Proceedings of the International Conference on Supercomputing*, ICS '18, New York, NY, USA, 2018. ACM.

[137] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, , Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, and Michael A Kozuch. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–493. IEEE, 2016.

[138] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. 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)*, pages 79–91, June 2015.

[139] Andre Seznec. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers*, 2004.

[140] Agam Shah. Micron's Revolutionary Hybrid Memory Cube Tech is 15 Times Faster than Today's DRAM, 2014.

[141] Yakun Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators. In *HPCA Sensors and Cloud Architectures Workshop (SCAW)*, 2015.

[142] Jaewoong Sim, Alaa R Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent hardware management of stacked dram as part of memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24. IEEE, 2014.

[143] Jaewoong Sim, Gabriel H. Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi. A mostly-clean dram cache for effective hit speculation and self-balancing dispatch. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 247–257, Washington, DC, USA, 2012. IEEE Computer Society.

[144] Avinash Sodani. Race to Exascale: Opportunities and Challenges, 2011.

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

[146] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. Bliss: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):3071–3087, Oct 2016.

[147] Madhusudhan Talluri and Mark D. Hill. 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, pages 171–182, New York, NY, USA, 1994. ACM.

[148] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[149] Vijay Tatkar. What is the sparc m7 data analytics accelerator? `https://community.oracle.com/docs/DOC-994842`. [Online; accessed 04-Aug-2018].

[150] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Hardware monitors for dynamic page migration. *J. Parallel Distrib. Comput.*, 68(9):1186–1200, September 2008.

[151] Linus Torvalds. Performance profiling on core kernel code. https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6, April 2014. [Online; accessed 31-Jul-2018].

[152] UEFI.org. Advanced Configuration and Power Interface Specification, Version 6.2. http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf, May 2017. [Online; accessed 31-Jul-2018].

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

[154] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349, Oct 2011.

[155] VMware. Performance Best Practices for VMware vSphere 5.0, 2011.

[156] Hao Wang, Jie Zhang, Sharmila Shridhar, Gieseo Park, Myoungsoo Jung, and Nam Sung Kim. Duang: Fast and lightweight page migration in asymmetric memory systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–493. IEEE, 2016.

[157] Andy Whitcroft. sparsemem memory model. https://lwn.net/Articles/134804/. [Online; accessed 04-Aug-2018].

[158] Wikipedia. Violin plot. https://en.wikipedia.org/wiki/Violin_plot. [Online; accessed 04-Aug-2018].

[159] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry G. Baler, editor, *Memory Management*, pages 1–116, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[160] Yuan Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE Des. Test*, 28(1):44–51, January 2011.

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

[162] Zi Yan. Accelerating page migrations. https://lwn.net/Articles/714991/. [Online; accessed 04-Aug-2018].

[163] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 430–443, New York, NY, USA, 2017. ACM.

[164] Jason Zebchuk, Babak Falsafi, and Andreas Moshovos. Multi-grain coherence directories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 359–370, New York, NY, USA, 2013. ACM.

[165] Ross Zwisler. Surface Heterogeneous Memory Performance Information. `https://lwn.net/Articles/727348/`, July 2017. [Online; accessed 31-Jul-2018].

URLs in references were last accessed December 13, 2018