IMPROVING AND COMPLEMENTING VIRTUAL MEMORY USING HARDWARE TECHNIQUES

by

GUILHERME MOTA CAVALCANTI DE ALBUQUERQUE COX

A dissertation submitted to the

School 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

October, 2018

ABSTRACT OF THE DISSERTATION

Improving and Complementing Virtual Memory Using Hardware Techniques

by GUILHERME MOTA CAVALCANTI DE ALBUQUERQUE COX

Dissertation Director: Abhishek Bhattacharjee

Virtual memory is a classic computer science abstraction and is ubiquitous in all scales of computing today. However, despite years of research, virtual memory faces critical performance and security challenges. This thesis aims to address these challenges. The first challenge we address is the growing performance overheads faced by virtual memory as workloads continue demanding ever-increasing amounts of memory. The key culprit of these overheads is address translation, the mechanism by which virtual memory translates a program's virtual addresses to physical addresses. Performing fast address translation requires the design of fast and efficient hardware translation lookaside buffer (TLB) caches. Unfortunately, TLBs struggle to perform efficiently for "big data" workloads. This thesis proposes a range of hardware mechanisms to improve TLB performance. The second challenge we address pertains to the security mechanisms offered by the virtual memory abstraction through memory protection and process isolation. Despite their utility, protection/isolation are insufficient to avoid important classes of remote attacks. Attackers can corrupt the operating system and thereby gain control of the entire machine. Therefore, there is a need for security mechanisms complementary to those provided by virtual memory. We propose low-overhead mechanisms achieve this. Our approach is to build hardware that can snapshot physical memory in an efficient manner, so that we can enable faster/better memory forensics to enhance system security.

Both sets of studies highlight some important themes in this thesis. One unifying theme of our work is to build hardware mechanisms that are transparent to application developers and systems programmers. Another unifying theme is ease of implementation – we deliberately use hardware mechanisms that require modest hardware modifications. In situations when the modifications are more substantial, we formally verify the correctness of our approach. Finally, we quantify the benefits of our approaches using not only software performance models (like most architecture studies), but also go beyond by quantifying real-system performance when possible.

Acknowledgements

A Ph.D. is a long journey where persistence and endurance are put to the test. Throughout my years in the program, I am happy to say that many things were accomplished beyond the degree. I have learned about how to research in Systems in Computer Science. I met an incredible amount of really smart people who helped me along the way. I made great friends, people that are going to be present in my life for the rest of it.

Chronologically, I must recognize and thank the support, the letters of recommendation, and the incentive given by Cristiana Bentes and Ricardo Farias, my Masters co-advisors in Brazil. They paved the way for me to be admitted into the Rutgers Ph.D. program. Along with them, my friend David Cluxton who played a fundamental role in my preparation for the entrance exams. Ricardo Bianchini, for having accepted me to the program and helped me find my research advisor.

I am grateful for the opportunity to learn from and interact with many faculty members from Rutgers, particularly Liviu Iftode, Santosh Nagarakatte, Thu Nguyen, Ulrich Kremer, and Vinod Ganapathy.

I would not have completed my Ph.D. without the guidance, support, and trust of my advisor Abhishek Bhattacharjee. The thorough, detailed, forward-looking way to brainstorm new ideas and projects has been vital to the success of my Ph.D. More than an advisor, we have become friends after so many heated and inspiring debates.

I was lucky to work with a group of motivated and bright fellow students. The members of the RUArch lab and the systems group at Rutgers: Bill Katsak, Binh Pham, Chris Woithe, Jae Woo Joo, Jan Vesely, Karthik Sriram, and Zi Yan. Members of the extinct DarkLab: Cheng Li, Inigo Goiri, Luiz Ramos, and Qingyuan Deng.

My wife Marina who joined me in this adventure, my daughter that unknowingly raised the bar by keeping me awake more than what one would expect, but also rewarding me every time that I came back home from the lab. My mother Marize and sisters Gabriela and Roberta, for all the remote, but important, support.

Finally, the support of friends and family, people that one way or another helped me cope with the challenges of the Ph.D.

Dedication

To my wife Marina, my daughter Olivia,

along with my family and friends

for their encouragement, support, and unconditional love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Virtual memory is a classic computer science abstraction that has been vital to the success of computing over several decades. It used ubiquitously today in systems as diverse in scale as mobile devices, wearable devices, desktops/laptops, and even large-scale data centers. Virtual memory's success can be attributed to benefits for programmers: **programmability** and **security**. The mechanisms used to provide these benefits are *address translation* and *memory protection*, which are both implemented by all modern page-based virtual memory systems. Even though it is possible to implement these features without page-based virtual memory, all modern systems pack them as a whole. This thesis shows the pitfalls, however, facing the traditional virtual memory abstraction. We show that both address translation and memory protection have become inefficient and incomplete in terms of providing the needs that modern systems have.

Virtual Memory				
	Programmability via Address Translation		**Security via Memory Protection**	
Trend	*Problem*	*Solution*	*Problem*	*Solution*
Heterogeneity	Unified VM for CPUs and accelerators	MIX TLBs (§2) NOCSTAR (§3) PTW-SCHED(§4)	Heterogeneity in the techniques and platforms used to enhance security	SNIPSNAP(§5)
Big data	TLB reach for ever-increasing memory sizes	MIX TLBs (§2) NOCSTAR (§3)	Security without compromising performance	SNIPSNAP(§5)

Figure 1.1: Thesis overview

Figure 1.1 illustrates the overall structure of this thesis, the problems that focus on, and our approaches to solving these problems. Specifically, we focus on:

Programmability:

- Virtual memory improves programmability by enabling programmers to view memory as a flat and linear array of bytes, thereby hiding the complexity of the physical memory which is made up of a complex assortment of memory and storage devices [77, 76, 36, 98]. This means, however, that program virtual addresses must be translated to physical addresses on all memory references - an operation known as address translation. The problem with address translation is that the structures that are used to accelerate it, such as Translation Lookaside Buffers (TLB), Memory Management Unit (MMU) caches, and Page Table Walkers (PTW), were built at a time when CPUs running single-threaded workloads were the de facto standard. Our world is very different today and these hardware structures have now suffer from performance problems [29, 66, 34]. Specifically, the advent of big data workloads means that CPUs as well has hardware accelerators (*e.g.,* GPUs, etc.) require infeasibly large TLBs for efficient translation. This is why, for example on CPUs, we find execution time overheads of up to 50% due to address translation [29, 66].

- While ensuring that the programmability benefits of virtual memory continue to be realized efficiently on CPUs is already challenging, these problems are even more pronounced when one considers hardware accelerators. Consider, for example, the GPU, which has become a key accelerator in the server, datacenter, cloud, and high-performance computing domains. GPUs also benefit from virtual memory when processing an ever-increasing set of general-purpose applications [13, 134]. However, implementing GPU virtual memory support is challenging because TLBs must be enormous to cope with the GPU's high levels of concurrency [175, 172, 195, 216, 24]. Consequently, GPU virtual memory overheads can slow down application runtime by as much as 3.7-4.0$\times$ [210, 195]. Such problems posed by heterogeneity are not restricted to GPUs alone – for example, large TLBs are particularly ill-sutied for area-constrained fixed function units and other accelerators.

The first problem this thesis addresses is the design of more efficient address translation hardware for CPUs and hardware accelerators like GPUs.

Security:

- One of the underpinnings of modern computer security is the memory protection facility provided by virtual memory [111, 77, 76, 36]. Nevertheless, while memory protection is crucial to security, is not sufficient in and of itself [51, 191]. One important class of security vulnerabilities – and a focus of our thesis – that cannot be obviated by protection alone is that of remote rootkit attacks compromising systems software in data-center and cloud environments. The core reason that virtual memory is insufficient for these types of attacks is that it was conceived at a time predating modern security attacks. But today, the simple time-sharing mainframes that virtual memory was design on have given way to data-center environments with complex rack-scale systems with disaggregated or distributed shared memory running many layers of sophisticated software. Therefore, we now rely on mechanisms beyond memory protection to give us the security guarantees modern systems need. Our goal in this thesis is to show how we can built hardware complementary to virtual memory to enable techniques like memory forensics [51, 191] in this dramatically different computing landscape.

- Unfortunately, modern memory forensics techniques suffer from performance problems posed by increasing amounts of memory used by workloads. In particular, memory forensics need to take a snapshot – a scan of a workload's entire virtual address space and/or the system physical address space. As memory grows, so does the time taken to snapshot memory. Therefore our goal is to use hardware techniques to acquire snapshots quicker than the time it takes today (*e.g.,* minutes in a typical server to snapshot its memory), during which the entire machine and its services are stalled.

The second problem this thesis addresses is the question of complementing traditional virtual memory security mechanisms with hardware for faster snapshots and better memory forensics techniques.

1.1 My Research Contributions

To detail solutions to the problems described above, I now provide a conceptual overview of the problems facing address translation and security.

Address Translation: The first topic in this thesis is faster address translation. We discuss two trends that make this challenging to achieve. The first trend is the advent of *heterogeneity*, or the integration of hardware accelerators. With heterogeneity, computer systems add accelerators on a chip. Writing code to exploit the benefits of these accelerators is complex [13, 134]. Additionally, accelerators have size, power, and performance constraints. To simplify the programming model with CPUs and accelerators, a single unified address space visible to all processing elements is desirable. The benefits of a unified address space are well established by recent studies [175, 172]. The benefits range from a pointer is a pointer semantics, *i.e.,* the idea that programmers can use the same data structure in any of the processing elements. Additionally, unified address spaces spare programmers from having to carefully managing data copies and data marshaling between processing elements. Providing a unified address space is one crucial step at the realization of a genuinely heterogeneous programming model. To achieve this, however, we need to implement efficient address translation hardware in all of our accelerators. This is problematic because many accelerators have tight area, energy, and power constraints. We explore the question of how to design efficient TLB for all of our accelerators, specifically in the context of programmable GPUs.

A related question that affects address translation is that of physical memory capacity. Along with increasing heterogeneity in processing, vendors are also adopting heterogeneity in memory. Ultimately, this means that the physical address space is growing fast to keep pace with the needs of big data workloads. This is the second trend motivating our work – *“big data”*. Unfortunately, big-data workloads necessitate larger TLBs to enable access to data efficiently. The questions that arise are: How can we scale our TLBs? How can we particularly scale our TLBs for area-constrained accelerators? These are the problems our work addresses in Chapters 2, 3, and 4.

Security: We use the same two trends to address the second problem of this thesis, which is to improve security on modern systems. We have already established that the VM memory protection is not sufficient to fully secure a modern computer system. The challenge is on how to enhance security for the type of heterogeneous systems that we discussed above, which may potentially run a range of operating systems, be connected to a vast variety of peripherals, etc. To take a step towards holistic security, we complement memory protection with memory forensics. Memory forensics is a branch of computer forensics that focuses on acquiring and analyzing all the data in memory. After these analyses, the system is deemed to be corrupted or secure. What is still an open problem is how to acquire the entire physical memory of a computer system in a secure, complete, consistent, and efficient manner. We explore this problem in this thesis, showing that our mechanism to snapshot the memory achieves all of these properties. All prior techniques can be categorized into two groups: non-consistent but fast snapshots, or consistent but slow snapshots. Non-consistent and fast snapshots do not halt the machine while acquiring the data. However, their non-consistency means that memory analysis tools may not be able to detect all manners of attacks. On the other hand, consistent snapshots overcome this problem, but with a pernicious performance penalty. This performance penalty is worsening because of *“big data”* trends. Our solution provides a highly efficient, secure and consistent memory snapshot; this is addressed in Chapter 5 of this thesis. We now detail our specific research contributions.

1.1.1 Efficient Address Translation for Multiple Page Sizes

In almost all modern systems today, processors and operating systems (OSes) support multiple memory page sizes. These systems have a base page size (or small page size), typically 4KBs, and page sizes that are multiples of the base page size. For example, x86-64 has support to 2MB and 1GB page sizes. A page that is larger than a base page size is called a superpage. Superpages increase Translation Lookaside Buffer (TLB) hits, while small pages provide fine-grained memory protection. Ideally, TLBs should perform well for any distribution of page sizes. In reality, set-associative TLBs - used frequently for their energy efficiency compared to fully-associative TLBs - cannot (easily) support multiple page sizes concurrently. Instead, commercial systems typically implement separate set-associative TLBs for different page sizes. This means that when superpages are allocated aggressively, TLB misses may, counter-intuitively, increase even if entries for small pages remain unused (and vice-versa). This happens because TLBs have fixed size that the operating systems are not aware of.

We propose MIX TLBs, energy-frugal set-associative structures that concurrently support all page sizes by exploiting superpage allocation patterns. MIX TLBs boost the performance (often by 10-30%) of big-memory applications on native CPUs, virtualized CPUs, and GPUs. MIX TLBs are simple and require no software changes.

1.1.2 Scalable Distributed Shared Last-Level TLBs

Modern computer systems implement per-core multi-level TLBs. Recent studies have, however, shown the potential of replacing private per-core L2 TLBs with a last-level TLBs shared by multiple cores. A key stumbling block hindering their effectiveness however is their high access time.

We present a design methodology to reduce these high access times so as to realize high-performance and *highly scalable* shared L2 TLBs. As a first step, we study the benefits of replacing monolithic shared TLBs with a distributed set of small TLB slices. While this approach does reduce TLB lookup latency, it increases interconnect delays in accessing remote slices, jeopardizing overall performance. Therefore, as a second step, we devise a lightweight single-cycle interconnect among the TLB slices by tailoring wires and switches to the unique communication characteristics of memory translation requests and responses. Our approach combines the high hit rates of shared TLBs with low access times of private L2 TLBs, enabling significant system performance benefits.

1.1.3 Scheduling Page Table Walks for Irregular GPU Applications

Throughput-oriented accelerators, such as GPUs, pose pressure on TLBs and page table walkers. "Big data" applications with memory accesses with poor locality can halt a GPU completely due to outstanding address translation requests. Recent studies [210] on commercial hardware demonstrated that irregular "big data" GPU applications can bottleneck on virtual-to-physical address translations. We explore ways to reduce address translation overheads for such applications.

We discover that the order of servicing a GPU's address translation requests (specifically, page table walks) plays a key role in determining the amount of translation overhead experienced by an application. We find that different SIMD instructions executed by an application require vastly different amounts of *work* to service their address translation needs, primarily depending upon the number of distinct pages they access. We show that better forward progress is achieved by prioritizing translation requests from the instructions that require less work to service their address translation needs.

Further, in the GPU's Single-Instruction-Multiple-Thread (SIMT) execution paradigm, *all* threads that execute in lockstep (wavefront) need to finish operating on their respective data elements (and thus, finish their address translations) before the execution moves ahead. Thus, batching walk requests originating from the same SIMD instruction could reduce unnecessary stalls. We demonstrate that the reordering of translation requests based on the above principles improves the performance of several irregular GPU applications by 30% on average.

1.1.4 Secure, Consistent, and Fast Memory Snapshotting

Many security and forensic analyses rely on the ability to fetch memory snapshots from a target machine. With that, we can complement the security mechanisms offered by VM memory protection. To date, the security community has relied on virtualization, external hardware or trusted hardware to obtain such snapshots. These techniques either sacrifice snapshot consistency or degrade the performance of applications executing atop the target. We present SnipSnap, a new snapshot acquisition system based on on-package DRAM technologies that offers snapshot consistency without excessively hurting the performance of the target's applications. We realize SnipSnap and evaluate its benefits using careful hardware emulation and software simulation.

1.2 Thesis Organization

Having described the problems this thesis addresses, Figure 1.1 also shows the various chapters of this thesis and how we address these problems. In more detail:

Chapter 2 describes MIX TLBs [66] (published in ASPLOS '17), which is used to achieve efficient address translation on CPUs and GPUs, in bare-metal and virtualized environments.

Chapter 3 describes Nocstar [31] (published in MICRO '18), which is used to achieve efficient address translation on CPUs with big-memory systems in bare-metal and virtualized environments.

Chapter 4 describes a novel GPU TLB miss scheduling [195] (published in ISCA '18), a mechanism to efficiently reorder page table walks of irregular GPU applications.

Chapter 5 describes SnipSnap [67] (published in CODASPY '18), a mechanism to efficiently and securily acquire memory snapshots by complementing the virtual memory abstraction.

Chapter 6 discusses conclusions and future directions.

Chapter 2

Efficient Address Translation for Multiple Page Sizes

2.1 Introduction

The operating system's (OS') choice of page sizes for an application's memory needs critically impacts system performance. Modern processors and OSes maintain multiple page sizes. Superpages (or large pages) increase Translation Lookaside Buffer (TLB) hit rates [158, 201, 202]. Small pages provide fine-grained page protection and permissions [158, 202, 170]. This work's objective is to design a TLB that leverages any distribution of page sizes, with the following properties:

① **Good performance**: TLB hardware should not be underutilized and conflict misses should be avoided.

② **Energy efficiency**: TLBs can consume a significant amount – as much as 13-15% [80, 124, 120, 121, 196] – of processor energy. Our design should be energy-efficient.

③ **Simple implementation**: TLBs reside in the timing-critical L1 datapath of pipelines, and must be simple to meet timing constraints. This means that TLB lookup, miss handling, and fill must not be complex.

Meeting all three objectives, while handling multiple page sizes, is challenging. Meeting ② means that we use set-associative rather than fully-associative TLBs. However, set-associative TLBs cannot (easily) support multiple page sizes. This is because, on lookup, they need the lower-order bits of the virtual page number to select a TLB set. But identifying the virtual page number requires the page size, so that the page offset bits can be masked off. This presents a chicken-and-egg problem, where the page

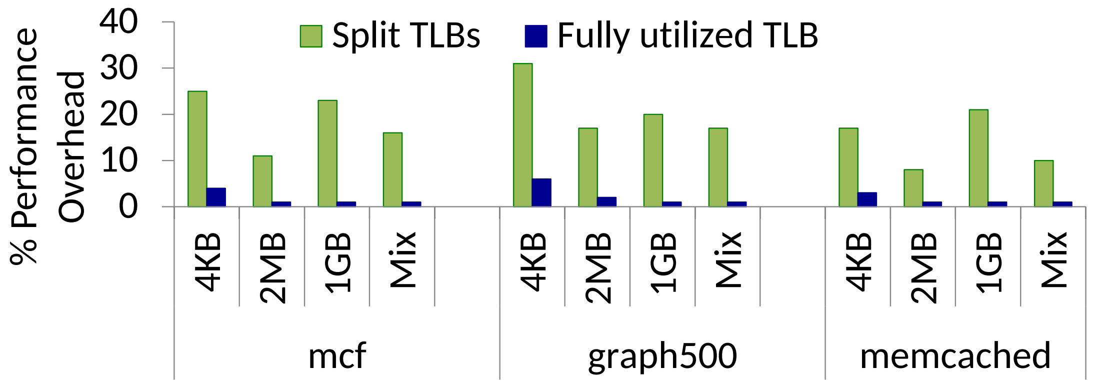


Figure 2.1: Percentage of runtime devoted to address translation, running natively on Intel Haswell with Linux (green). We assume cases where the OS allocates only one page size (4KB, 2MB, 1GB) and when page sizes are mixed. We compare performance against an ideal case where all TLB resources are well-utilized (blue).

size is needed for TLB lookup, but lookup is needed to determine page size. In general, industry and academia have responded in two ways, which compromise ① and/or ③.

Split TLBs: Most processor vendors use split (or partitioned) TLBs, one for each page size [109, 110, 161]. This side-steps the need for page size on lookup. A virtual address can look up all TLBs in parallel. Separate index bits are used for each TLB, based on the page size it supports; *e.g.,* the set indices for split 16-set TLBs for 4KB, 2MB, and 1GB pages (assuming an x86 architecture) are bits 15-12, 24-21, and 33-30 respectively. Two scenarios are possible. In the first, there is either hit in one of the split TLBs, implicitly indicating the translation's page size. In the second, all TLBs miss [161].

Unfortunately, while split TLBs achieve ③, and arguably ②, they often underutilize TLBs and compromise ①. The problem is that if the OS allocates mostly small pages, superpage TLBs remain wasted. On the other hand, when OSes allocate mostly superpages, performance is (counterintuitively) worsened because superpage TLBs thrash while small page TLBs lie unused [84, 48]. Figure 2.1 quantifies the extent of the problem, showing the percentage of runtime that mcf, graph500, and memcached devote to address translation. Results are collected using performance counters on Intel Haswell systems with 84GB of memory, running Linux with the methodology of Section 2.6. We assume that the OS allocates only a fixed page size (*i.e.,* 4KB, 2MB, 1GB) or mixed pages. One would expect that using large pages consistently improves performance. In reality, performance remains poor even with, for example, 1GB pages (green bars). Further, we compare these numbers to a hypothetical ideal set-associative TLB which can support all page sizes (blue); the gap with the green bars indicates the performance potential lost due to poor utilization of split TLBs.

Multi-indexing approaches: In response to this problem, past work has augmented set-associative TLBs to concurrently support multiple page sizes [161, 193]. Unfortunately, while this does improve ①, it does so at the cost of ② and ③. The central problems, described in Section 2.5.1, are variable access latencies, increased access energy, and complex implementation. Even in the rare cases when they are implemented commercially, they don't support all page sizes (*e.g.,* Intel's Haswell, Broadwell, and Skylake L2 TLBs cache 4KB and 2MB pages together but not 1GB pages, which require separate TLBs [109, 110]).

Our contributions: This work proposes (**MIX**) **TLBs**, fast ①, energy-efficient ②, and readily-implementable ③ structures that concurrently support all page sizes. MIX TLBs use a single set-indexing scheme – the one for small pages (*e.g.,* 4KB pages on x86) – for translations of all page sizes. While this simplifies the design, it also presents a problem. We use bits within the superpage page offset to select a TLB set. This means that a superpage is mapped to multiple (potentially all) TLB sets, an operation we refer to as **mirroring** (see Section 2.3). We overcome this problem, however, by observing that OSes frequently (though they don't have to) allocate superpages (not just their constituent small pages) in adjacent or *contiguous* virtual and physical addresses. We detect these adjacent superpages, and **coalesce** them into the same TLB entry (see Section 2.3). If we coalesce as many, or close to as many, superpages as the number of mirror copies – which we usually can in real-world systems – we counteract the redundancy of mirrors, achieving energy-efficient performance.

This work showcases MIX TLBs, their ease of implementation, and performance improvements of 10-30%. Using real-system characterization and careful simulation, we compare MIX TLBs to traditional set-associative designs, and previously proposed TLBs for concurrent page sizes [161, 193]. We also characterize superpage allocation patterns. Our results focus on Linux, but we've also studied FreeBSD and Solaris. One might initially expect that highly loaded sytems with long uptimes would be hard-pressed to defragment memory sufficiently to allocate superpages adjacently. Indeed, we observe that if system memory is sufficiently fragmented, OSes rarely generate superpages at all. However, we also observe that if OSes can generate even a few superpages, they have usually defragmented memory sufficiently to generate other adjacent and contiguous superpages too. MIX TLBs outperform their counterparts in both cases. When superpages are scarce, MIX TLBs use all TLB resources for small pages. When superpages are present, MIX TLBs seamlessly leverage any distribution of page sizes.

2.2 Scope of This Work

Systems are embracing workloads with increasing memory needs and poorer access locality (*e.g.,* massive key-value stores, graph processing, data analytics, deep learning frameworks, etc.). These workloads stress hardware TLB performance; as a result, address translation overheads often consume 15-30% of runtime today [169, 168, 29, 33, 35].

MIX TLBs also aid virtualized systems, where address translation is even more pernicious. Virtualized systems require two dimensions of address translation - guest virtual pages are converted to guest physical pages, which are then translated to system physical pages [170, 84, 32]. Two-dimensional page table walks are expensive, requiring 24 sequential memory accesses in x86 systems, instead of the customary 4 accesses for non-virtualized systems. Virtualization vendors like VMware identify TLB misses as a major culprit in the performance difference between non-virtualized and virtualized systems [170, 84, 48].

Finally, vendors have begun embracing shared virtual memory abstractions for heterogeneous systems made up of CPUs and GPUs [172, 173, 175, 17, 18, 134, 210, 216], accessing a single virtual address space. This allows "a pointer is a pointer everywhere" simplications of the programming model [134, 210]. However, now GPUs must also

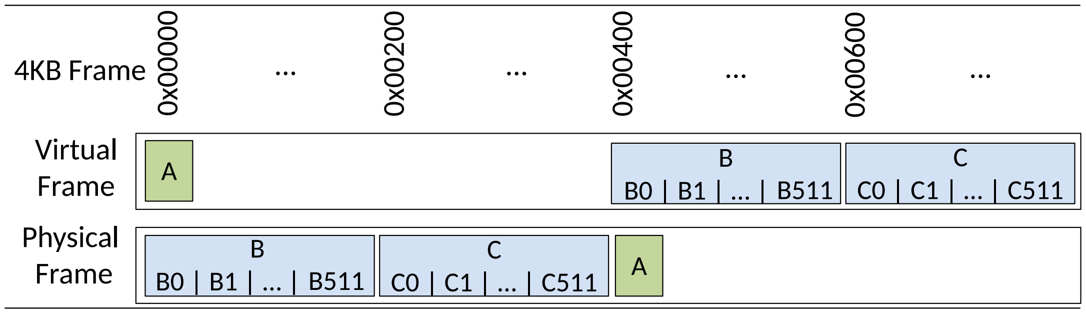


Figure 2.2: Example address space in an x86-64 architecture. We show 4KB frame numbers in hexadecimal. For example, translation B is for a 2MB page, made up of 4KB frame numbers B0-B511. 2MB translations B-C are contiguous.

perform address translation, just like CPUs. GPU TLBs are critical to performance as they must service the demands of hundreds to thousands of concurrent threads [172, 173, 210]. Unfortunately, we find that CPU-GPU systems also suffer from TLB utilization issues when using multiple page sizes.

2.3 High-Level Approach

We compare MIX TLBs to traditional split TLBs, using the address space of Figure 2.2. We show virtual and physical address spaces, with translations for small pages (A), and superpages (B-C). Without loss of generality, we assume an x86-64 architecture with 4KB and 2MB pages (1GB are handled similarly). Note that while we assume 64-bit systems, our examples show 32-bit addresses to save space. These addresses are shown in 4KB frame numbers (full addresses can be constructed by appending 0x000). Therefore, superpage B is located at virtual address 0x00400000 and physical address 0x00000000. Superpages B and C have 512 constituent 4KB frames, indicated by B0-511 and C0-511.

Figure 2.3 illustrates the lookup and fill operation of MIX TLBs and contrasts it to split TLBs. In step ①, B is looked up. However, since B is absent (both split and MIX TLBs maintain only A), the hardware page table walker is invoked ②. The page table walker reads the page table in units of caches lines; since a typical cache line is 64 bytes, and translations are 8 bytes, 8 translations (including B and C) are read in the cache line. Split TLBs then fill B into the superpage TLB ③. Unfortunately, there remains

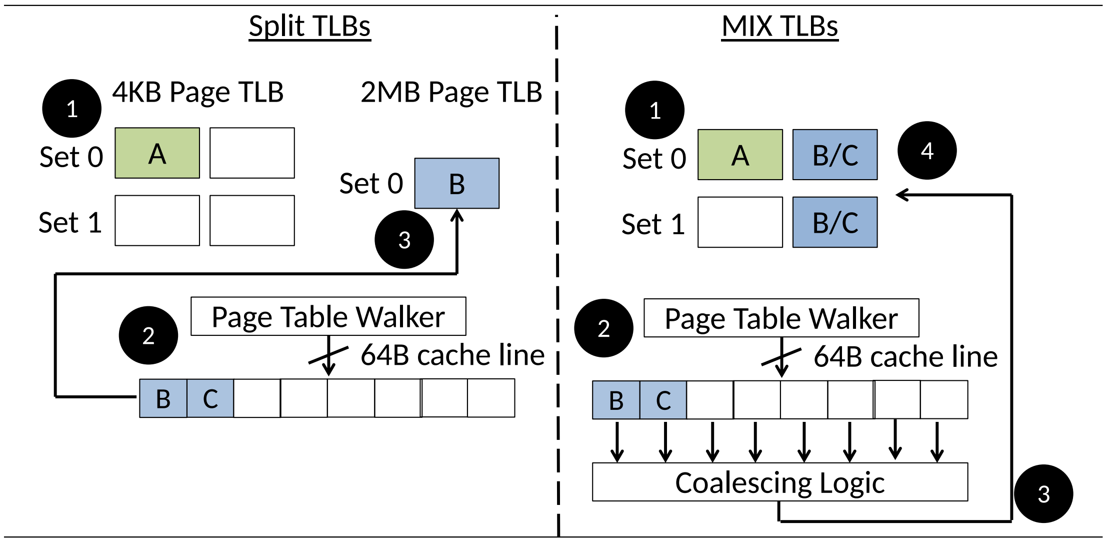


Figure 2.3: Superpage B lookup and fill for split versus MIX TLBs.

no room for C despite 3 unused small page TLB entries.

MIX TLBs, on the other hand, cache all page sizes. After a miss ① and a page table walk ②, we must fill B in the correct set. This presents a challenge; since MIX TLBs use the index bits for small pages (in our 2-set TLB example, bit 12) on all translations, the index bits are picked from the superpage's page offset. Thus, superpages do not uniquely map to either set. Instead, we mirror B in both TLB sets.

Mirroring presents a problem. Whereas split TLBs maintain one copy of a superpage translation, MIX TLBs maintain several mirror copies, reducing effective TLB capacity. However, MIX TLBs counteract this problem with the following observation – OSes frequently (though they don't have to) allocate superpages adjacently in virtual and physical addresses. For example, Figure 2.2 shows that B and C are contiguous, not just in terms of their constituent 4KB frames (*e.g.,* B0-511 and C0-511) but also in terms of the full superpages themselves. MIX TLBs exploit this contiguity; when page table walkers read a cache line of translations ②, adjacent translations in the cache line are scanned to detect contiguous superpages. We propose, similar to past work [169, 168], simple combinational coalescing logic for this ③. In our example, B and C are contiguous and are hence coalesced and mirrored. Coalescing counteracts mirroring. If there are as many contiguous superpages as there are mirror copies (or close to as

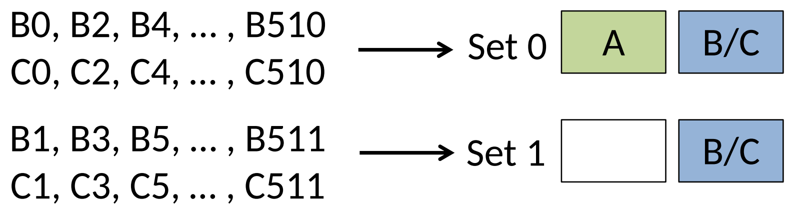


Figure 2.4: Though superpages B and C are maintained by multiple sets but on lookup, we only probe the set corresponding to the 4KB region within the superpage that the request is to.

many), MIX TLBs coalesce them to achieve a net capacity corresponding to the capacity of superpages, despite mirroring.

Crucially, Figure 2.4 shows that MIX TLB lookup remains simple. While coalesced mirrors of superpages reside in multiple sets, lookups only probe one TLB set. In other words, virtual address bit 12 in our example determines whether we are accessing the even- or odd-numbered 4KB regions within a superpage; therefore accesses to B0, B2, etc., and C0, C2, etc., are routed to set 0.

Naturally, this overview presents several important questions. We briefly address them below:

Why do MIX TLBs use the index bits corresponding to the small pages? Specifically, one may instead consider using the index bits corresponding to the superpage and apply that on small pages too. In our example, this would be like using virtual address bit 21 as the index (assuming we base the index on 2MB superpages). The advantage of this approach is that each superpage maps uniquely to a set, *eliminating* the need for mirrors (*e.g.,* B maps to set 0, and C maps to set 1).

Unfortunately, this causes a different problem. Now, spatially adjacent small pages map to the same set. For example, if we use the index bits corresponding to a 2MB superpage (*i.e.,* in our 2-set TLB example, bit 21), groups of 512 adjacent 4KB virtual pages map to the same set. Since real-world programs exhibit spatial locality, this elevates TLB conflicts (unless associativity exceeds 512, which is far higher the 4-8 way associativity used today [109, 110]). One could envision coalescing these small pages if the OS does allocate them contiguously in virtual and physical addresses; however past work shows that while small pages can be contiguous, they usually are not contiguous beyond more than tens of pages [169, 168]. We have evaluated using superpage index bits and have found that they increase TLB misses by 4-8$\times$ on average, compared to using small page index bits.

Why do MIX TLBs perform well? MIX TLBs are well utilized for any distribution of page sizes. When the system is highly fragmented and superpages are scarce, all TLB resources can be used for small pages. When the OS can generate superpages, it usually sufficiently defragments physical memory to allocate superpages adjacently too. MIX TLBs utilize all hardware resources to coalesce these superpages.

How many mirrors can a superpage produce and how much contiguity is needed? Assume that the superpage has N 4KB regions, and that our MIX TLB has M sets. N is 512 and 262144 for 2MB and 1GB superpages. Practical commercial L1 and L2 TLBs tend to have 16-128 sets [161, 109, 110]. Therefore, today's systems have N $>$ M, meaning that a superpage has a mirror per set (or N mirrors). However, if future systems see N $<$ M, there would be M mirrors.

Ultimately, good MIX TLB utilization relies on superpage contiguity. If the number of contiguous superpages is equal (or sufficiently near) the mirror count, performance is good. On modern 16-128 set TLBs, we desire (close to) 16-128 contiguous superpages. Section 2.7.1 shows that real systems do frequently see this much superpage contiguity. Section 2.4 shows how we can coalesce these many contiguous superpages, despite only scanning for contiguity within a single cache line, which maintain 8 translations, on a TLB miss.

2.4 Hardware Details

We now detail MIX TLB hardware, implementing them differently for the L1 and L2 levels. L1 MIX TLBs must be simple and fast; we sacrifice some coalescing opportunity to meet these requirements. L2 MIX TLBs can tolerate higher access latencies (*e.g.,* Intel and AMD L2 TLBs usually have 5-7 cycle access times [161]). Therefore, L2 MIX TLBs support more coalescing with (slightly) more complex hardware. MIX TLBs require no

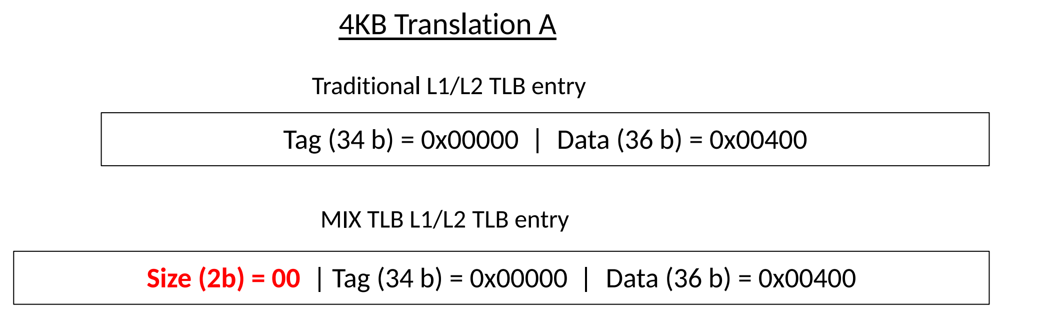

4KB Translation A
Traditional L1/L2 TLB entry
Tag (34 b) = 0x00000 | Data (36 b) = 0x00400
MIX TLB L1/L2 TLB entry
Size (2b) = 00 | Tag (34 b) = 0x00000 | Data (36 b) = 0x00400

Figure 2.5: Traditional TLB and MIX TLB entries for the translation corresponding to 4KB page A. We show the TLB entries at the L1 and L2 level, assuming both have 4 sets. MIX TLBs require just an additional 2 bits to record the page size.

OS or application changes.

2.4.1 MIX TLB Entries

MIX TLB entries are similar to traditional set-associative entries. We detail the modest differences between the two. Although actual x86-64 architectures can use up to 52-bit physical addresses, use assume 48-bit physical addresses in our example for simplicity. Extending this approach to 52-bits parallels our example.

Small pages: Figure 2.5 contrasts traditional TLB and MIX TLB entries for 4KB pages. We use translation A from Figure 2.2, and assume 4-set L1 and L2 TLBs. Therefore, the two least significant bits of the virtual page need not be stored in the tag. MIX TLBs only require a 2-bit page size field to distinguish among the 3 page sizes. Though they are not shown, the entries also maintain page permission bits.

Superpages: Figure 2.6 compares traditional to MIX TLB entries for superpages, assuming 2-set TLBs. Aside from the page size, MIX TLBs must maintain information about coalesced superpages. L1 entries use a bitmap for this. 2-set MIX TLBs maintain a 2-bit bitmap to record coalescing information of up to two superpages. Furthermore, since this entry caches superpage information, it uses 9 fewer tag bits for this versus small page entries. In fact, we can even drop a 10th bit because 2-bit bitmaps implicitly store information about 2×2MB (4MB) memory regions. These bits can be repurposed

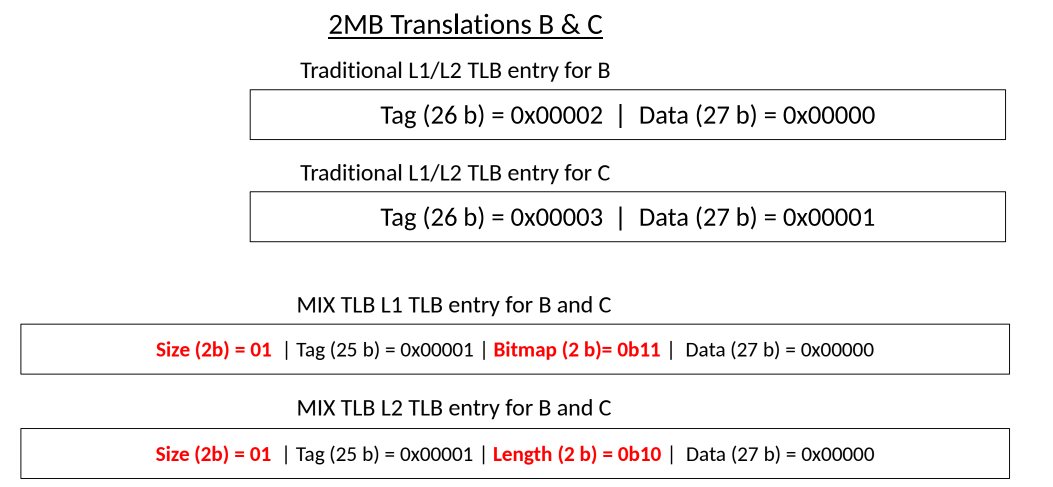

2MB Translations B & C
Traditional L1/L2 TLB entry for B
Tag (26 b) = 0x00002 | Data (27 b) = 0x00000
Traditional L1/L2 TLB entry for C
Tag (26 b) = 0x00003 | Data (27 b) = 0x00001
MIX TLB L1 TLB entry for B and C
Size (2b) = 01 | Tag (25 b) = 0x00001 | Bitmap (2 b)= 0b11 | Data (27 b) = 0x00000
MIX TLB L2 TLB entry for B and C
Size (2b) = 01 | Tag (25 b) = 0x00001 | Length (2 b) = 0b10 | Data (27 b) = 0x00000

Figure 2.6: Traditional TLB and MIX TLB entries for the translation corresponding to 2MB pages B-C. L1 MIX TLB and L2 MIX TLB entries use a bitmap and a length field to record contiguous superpages, respectively. We assume 2-set TLBs.

for the bitmap. Figure 2.6 records 0b11 to indicate information about contiguous superpages B and C.

L2 MIX TLBs record longer contiguity, with marginally greater complexity. Instead of a bitmap, we use a contiguity length field. Therefore, a 2-bit length field (though it could use more bits) records contiguity of up to 4 superpages.

MIX TLBs are only marginally bigger than a standard set-associative entry since the bitmap and length fields are repurposed with unused tag bits. Only a 2-bit page size field is added, increasing per-entry size by less than 1%.

Alignment restrictions: To simplify MIX TLB hardware, we only coalesce superpages that are suitably aligned. Specifically, to coalesce up to N superpages, only contiguous superpages that begin at virtual address boundaries of N may be coalesced. Our MIX TLB example in Figure 2.6, which coalesces up to 2 superpages, therefore only coalesces superpages that begin at multiples of 2×2MB or 4MB. This does reduce coalescing opportunity slightly, but as we show in Section 2.7.2, performance continues to be good.

Bitmap versus length: For the same number of bits, length fields record more information, allowing L2 MIX TLBs to coalesce longer runs of contiguous superpages. The

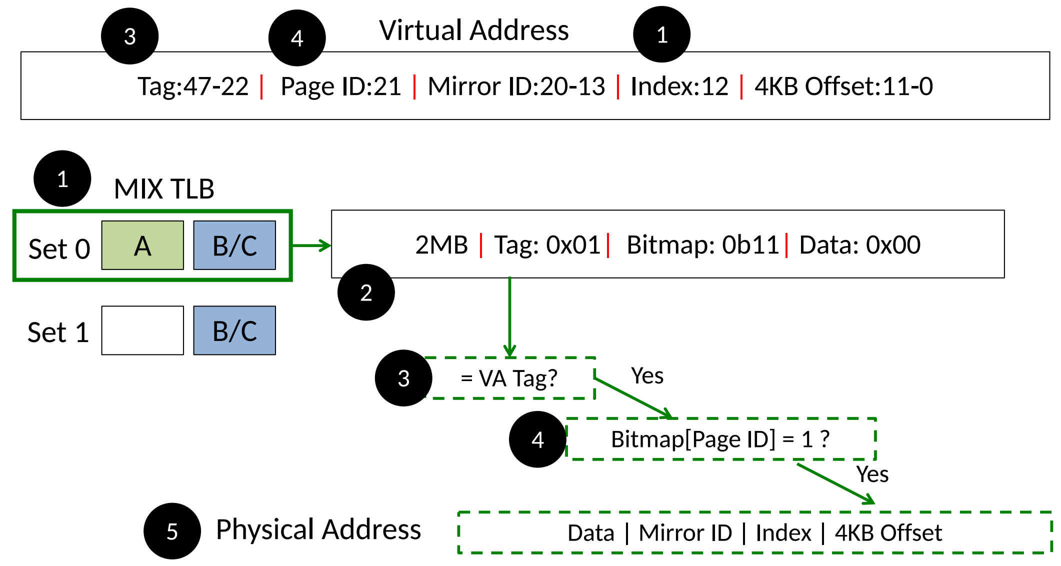

Virtual Address
3
4
1
Tag:47-22 | Page ID:21 | Mirror ID:20-13 | Index:12 | 4KB Offset:11-0
1
MIX TLB
Set 0
A
B/C
Set 1
B/C
2MB | Tag: 0x01 | Bitmap: 0b11 | Data: 0x00
2
3
= VA Tag?
Yes
4
Bitmap[Page ID] = 1 ?
Yes
5
Physical Address
Data | Mirror ID | Index | 4KB Offset

Figure 2.7: L1 MIX TLB lookup and hit (assuming a 2-set TLB). The physical address is found using bit shifting and concatenation.

slight downside is the slightly more complex TLB lookup this prompts (which we detail later in this section). L1 bitmaps do have one more advantage – they can record information about "holes" in contiguously allocated superpages.

2.4.2 MIX TLB Operation

In this section, we describe MIX TLB operation, including hits, misses, and fills.

L1 lookup: Figure 2.7 shows how L1 MIX TLBs are looked up. Since 4KB page lookups remain unchanged, we focus on superpages. Index bits are selected from the virtual address as per small page size – therefore, assuming a 2-set TLB and 4KB small pages, we use bit 12 as the index. Consequently, there is a question as to what happens with the remainder of the 2MB page offset, bits 20-13, and bits 11-0. We call bits 20-13 the mirror ID, as they identify individual 4KB regions within a superpage (*i.e.*, B0, B1, B2, etc., in Figure 2.4). Bits 11-0 are the offset within these 4KB regions. Finally, the remaining upper order bits of the virtual address are split into a tag and a page ID. The page ID identifies the specific superpage within a contiguous bundle – since our example assumes a 2-set TLB that can coalesce up to 2 entries, 1 page ID bit suffices to identify the desired superpage.

The index bits identify the MIX TLB set ①. In our example, we cache A and B-C in the set, so both entries are checked in parallel. The page size determines whether the entry is a coalesced superpage bundle ②. Then the tag is compared to the virtual address tag ③. If there is a match ④, the L1 bitmap must be checked to determine whether this superpage exists in the coalesced entry. This is accomplished by indexing the bitmap using the page ID; in our example with B-C, this is set. Therefore, the physical address can be constructed by concatenating the relevant fields of the lookup virtual address with the data field in the MIX TLB entry ⑤.

Note that this process essentially leaves lookup latency unchanged from the conventional TLB because it relies purely on bit shifts and concatenations.

L2 lookup: For L2 MIX TLB lookups, instead of checking a bitmap, the length field is checked against comparators to determine whether the desired virtual page falls within in the range of coalesced translations. Range matches (and their implementation) are well studied [169, 123]. On a range (and hence TLB) hit, the hardware computes the offset of the lookup virtual page against the tag information stored. This offset is added to the TLB entry's data field, calculating the desired physical address, similar to recent work [169, 123]. This does increase the L2 lookup latency; however we've modeled the hardware (see Section 2.6) and found that there is only a slight (*i.e.,* 3% increase) in lookup time.

Miss and fill: Section 2.3 sketched how MIX TLB are filled. By scanning for contiguous superpages and coalescing only on TLB misses, the coalescing logic is placed off the critical path of lookup. Therefore, it has low overhead, can be designed with simple combinational logic, and adds negligible latency or energy over a baseline set-associative TLB.

Prefetching and capacity strategies: Superpages provide two benefits. First, they record information about a far bigger portion of memory than small pages; hence, they reduce TLB conflict misses. However, they also provide prefetching benefits. For example, a TLB fill of a 2MB superpage obviates the need for 512 separate fills for 4KB pages.

This observation informs the way MIX TLBs coalesce. At first blush, one might consider only filling superpage information into the set that was probed by the lookup virtual address. For example, suppose that in Figure 2.4, there is a lookup for superpage B, but to the B0 region specifically. On a TLB miss, one option might be to only fill superpage information into set 0. To be sure, this does capture some of the prefetching benefits of superpages (*i.e.,* information for B2, B4, etc., are also filled), but it also loses prefetching potential for some 4KB regions (*i.e.,* B1, B3, etc.). For this reason, we instead fill as many sets as necessary with superpage mirrors to capture information about the full superpage.

In fact, our notion of prefetching goes beyond the prefetching benefits of superpages, because coalescing actually prefetches contiguous superpages around the requested superpage. To do this without complicating the page table walker, we prefetch (by coalescing) only contiguous superpages that sit in the same cache line as the page table translation (up to 8 superpages, see Figure 2.3). Note, however, that commercial Sandybridge/Haswell TLBs maintain 16-128 sets; this means that MIX TLBs should try to coalesce 16-128 superpages to offset mirroring. One needs to scan additional cache lines containing the page table to do this. Instead, we choose a simpler approach. We initially coalesce up to 8 superpage entries. When future memory references touch superpages adjacent to these coalesced entries, sitting in other cache lines, we detect this behavior and coalesce them into the existing MIX TLB entry. This achieves good performance in practice.

2.4.3 Interactions with Replacement Policies

One issue with MIX TLBs is that information for the same superpages are now distributed among multiple TLB sets. Figure 2.8 illustrates the challenges this brings up. We show a 2-set, 4-entry MIX TLB, explicitly indicating each set's LRU chain. Suppose that initially ①, it stores information about A and B-C. Now, requests for D and E, small pages mapping to set 1, arrive and are filled into the TLB in steps ② and ③. At this point, set 1's mirror copy of B-C is evicted, while set 0's copy remains. This

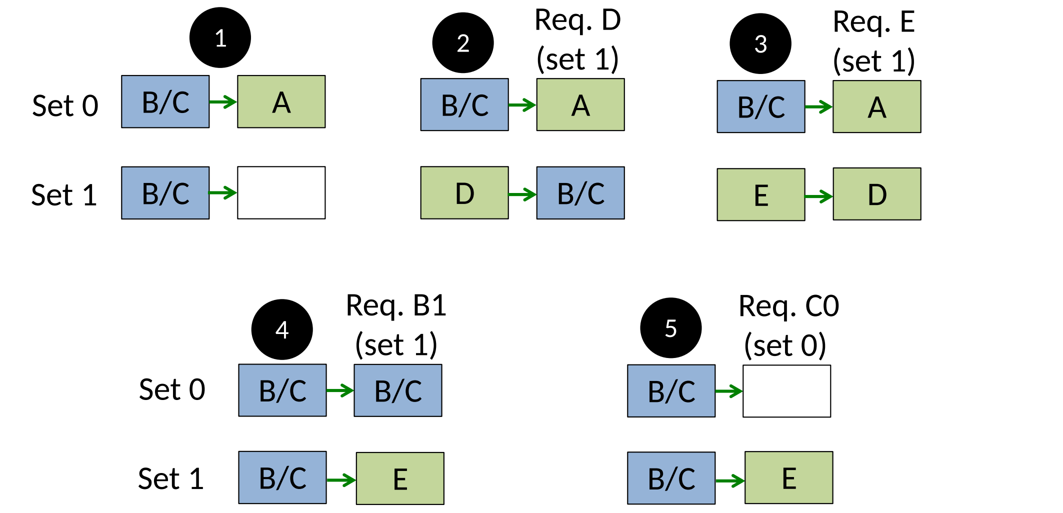


Figure 2.8: Replacement decisions are made independently on mirror copies, which can cause duplication issues.

presents a problem in ④, when we have a request for superpage B but in 4KB region B1, which maps to set 1. We see a TLB miss, and walk the page table; however, once we locate B-C, an important question is whether to mirror B-C into the other sets. On the one hand, the other sets may already have copies of B-C and blindly mirroring leads to duplicate copies. On the other hand, a TLB maintains 64-128 sets; scanning all the sets to check for duplicates is an energy-expensive and impractical approach. Therefore, we adopt the first approach; ④ shows that this leads to a duplicate B-C copy in set 0, evicting A. However, we can mitigate this problem when set 0 is probed in the future ⑤. Since all the entries in the set are checked for a tag match, we identify duplicates and eliminate copies.

2.4.4 OS Operations

Invalidations: The OS may change page table mappings through program execution and corresponding TLB entries must be invalidated. For small pages, this is achieved in the same way as conventional TLBs. For superpages, this is accomplished in L1 MIX TLBs by resetting the bitmap bit of the superpage in question. This permits superpages adjacent to the invalidated superpage to remain cached.

On L2 MIX TLBs however, this is slightly more complicated, because they maintain a length field. The simplest approach is to invalidate the entry corresponding to the entire coalesced bundle. A more sophisticated approach might split the entry into two separate entries around the invalidated translation. Since we find, in practice, relatively few invalidations, we take the (slightly) lower performance but simpler approach of invalidating the entire coalesced entry.

Permission bits: An important question is whether to coalesce adjacent superpages that use different access permission bits. While this could be accomplished with more storage in the MIX TLB to record differing permissions, we take the simpler (but high-performance) approach of only coalescing superpages when they have the same permission bits.

Dirty and access bits: Translations in page tables maintain access and dirty bits to aid the OS' page replacement policy. In some (though not all) architectures, like x86 and ARM, these bits are set by hardware, and read by the OS.

The x86 architecture mandates that only translations with access bits set to 1 in the corresponding page table entry may be filled into the TLB [111]. MIX TLBs therefore coalesce only translations with access bits set to 1 on TLB fill. Naturally, this does not preclude translations from being added to existing coalesced entries in the TLB once they are demanded by the processor and have their access bits set.

Page table entries also maintain a dirty bit, recording whether the page has been written to. Conventional TLBs maintain a dirty bit per entry; on a store instruction to the translation in the entry, this bit is checked. If the bit is 0, the hardware page table walker injects a micro-op instruction to write the software page table entry's dirty bit [111, 146, 181]. If the bit is 1, such updates are not needed.

On the one hand, MIX TLBs could maintain a dirty bit per superpage in a coalesced bundle. However, MIX TLBs support 16-128 superpages per coalesced bundle; requiring 16-128 dirty bits per TLB entry requires infeasible storage. On the other hand, we could require that only superpages with the same dirty bit value be coalesced; unfortunately, we've found that this drastically reduces coalescing opportunity.

Instead, our approach sets the MIX TLB entry dirty bit only if all the superpages in a coalesced bundle are dirty. If a single one of them is not dirty, the TLB entry's dirty bit is cleared. In practice, this means that every time there is a TLB miss and page table lookup, we check to see if the requested PTE's dirty bit is set. If it is clear, and the MIX TLB entry where this translation is to be coalesced has its dirty bit cleared (if it is not already clear). If it is set, the dirty bit of the MIX TLB entry is left unchanged. That is, if it was already dirty, it is left dirty again. Naturally, this approach adds cache traffic versus a scenario where the TLB has a dirty bit per translation. In practice though, we've found that performance remains good, and area overheads are modest.

2.4.5 Hardware and Energy Complexity

MIX TLBs are readily-implementable. As detailed, the size of each MIX TLB entry is roughly 1% bigger than a standard set-associative TLB. We've modeled these overheads using CACTI [154], and find that lookup latency and energy remains unchanged. MIX TLB misses do invoke coalescing logic; however, like prior work [169, 168, 33], we find that this requires only simple combinational logic. While it does impose (slight) delay overheads on TLB fill, we have modeled these overheads in RTL and find that they do not affect overall performance. Furthermore, while it is true that coalescing logic uses some area, MIX TLBs eliminate the need for separate superpage TLBs. Therefore, we ultimately *save* area.

Finally, we consider the energy costs of mirroring. Mirroring (coalesced) superpages into multiple sets does consume more energy than conventional TLBs, which fill one set. Modeling this using CACTI and RTL, however, we find that the much higher hit rates offered from MIX TLBs greatly reduce memory and cache references (for page table walks) and reduce runtime. Ultimately, the resulting energy benefits outweigh the energy overheads of mirroring.

2.5 Comparison to Past Work

MIX TLBs present a counterpoint to split set-associative TLBs. However, prior work has looked at alternate ways to provide mixed page support too. We now detail this past work, showing why MIX TLBs are superior.

2.5.1 Multi-Indexing Methods

Recent approaches to tackling the inadequacies of split set-associative TLBs can be summarized into three categories:

Hash-rehashing: We initially perform a TLB lookup (hash) assuming a particular page size (usually the baseline page size). On a miss, the TLB is again probed (rehash), using another page size. This continues until all page sizes are exhausted [161]. There are several drawbacks to this approach. TLB hits have variable latency, and can be difficult to manage in the timing-critical L1 datapath of modern CPUs [30], while TLB misses take longer. One could parallelize the lookups but this adds lookup energy, and complicates port utilization. Consequently, hash-rehashing approaches are used in only a few architectures, and that too, to support only a few page sizes (*e.g.,* Intel Skylake and Haswell architectures support 4KB and 2MB pages with this approach but not 1GB pages).

Skewing: Skewed TLBs are inspired by skewed associative caches [193, 192, 46]. A virtual address is presented to multiple parallel hashing functions. The functions are chosen so that if a group of translations conflict on one way, they conflict with a different group on other ways. Translations of different page sizes reside in different sets [193]. For example, if our TLB supports 3 page sizes, each cacheable in 2 separate ways, we need a 6-way skew-associative TLB.

Skew associative TLBs can be effective but also have problems. Lookups expend high energy as they require parallel reads equal to the sum of the associativities of all supported page sizes. Saving energy by reducing the associativity of page sizes decreases performance. Further, even the simplest skewing functions are usually not appropriate for latency-sensitive L1 TLBs [186, 185]. Finally, good TLB hit rates require effective replacement policies; unfortunately, skewing breaks traditional notions of set-associativity and requires complicated replacement decisions. In practice, skewed TLBs use area- and energy-expensive timestamps for replacement [186, 185]. Because of these problems, we know of no commercial skew-associative TLBs.

Prediction-based enhancements: Recent work [161] enhances hash-rehashing and skewing by using a hardware predictor to accurately guess the page size of a requested translation before TLB lookup. The hash-rehash or skew TLB is first looked up with this predicted page size; only on misses are the other page sizes used. When prediction is accurate, this approach lowers the average TLB hit latency and lookup energy, by first looking up with the "correct" page size. Problems remain with this approach. Predictors become complex as the number of page sizes increase. Further, predictors increase access latency variability since we now also have different latencies for hits with correct prediction, eventual hits after a wrong prediction, etc.

Overall, multi-indexing is complex, latency-variable, and can be energy-intensive. It is generally unsuitable for L1 TLBs. Even when implemented, it can scale poorly as the number of page sizes increase. In addition multi-indexing potentially complicates operations like TLB shootdowns, selective invalidations of global versus local translations, managing locked translations, etc., because of their multi-step lookup [161]. Instead, since MIX TLBs remain largely unchanged in implementation compared to standard set-associative TLBs, they do not suffer from these issues.

2.5.2 Prior Work on Page Allocation Contiguity

The concept of exploiting OS page allocation for better TLB performance has received recent attention [201, 169, 168, 29, 33, 30]. COLT [169] shows that small pages are often allocated contiguously in virtual and physical memory. These contiguous small pages are coalesced into 4KB TLBs for better performance. This builds on earlier work [201] which exploited certain patterns of page allocation contiguity and alignment.

We make two observations about MIX TLBs and their relationship with COLT. First, they solve a problem that COLT cannot – realizing a single set-associative TLB to cache multiple page sizes. Second, COLT observes that there are cases when small pages are allocated more frequently than superpages. In these cases, coalescing small pages helps TLB performance. Our work also observes that there are situations where superpages are hard to allocate. We provide orthogonal benefits to COLT in these cases, by utilizing TLB entries that would otherwise have been devoted only to superpages. In addition, unlike COLT, we also help performance in the numerous cases when superpages abound. Indeed, for these reasons, COLT can actually be combined with MIX TLBs (see Section 2.7.2).

2.6 Methodology

We use a mix of real-system measurements, memory tracing, and detailed simulation. We now describe these approaches.

2.6.1 Real-System CPU Measurements

To assess real-system split TLB performance and OS page allocation patterns, we use a dual-socket Intel processor with 4-way set-associative split TLBs for 4KB pages (64 entries) and 2MB pages (32 entries). 1GB L1 TLBs are fully-associative and 4 entries. We use a 512-entry L2 TLB for 4KB and 2MB pages, but not 1GB pages. Instead, there is a separate 32-entry L2 TLB for 1GB pages. Further, this system is equipped with a 24MB LLC and 80GB of memory.

We focus on Linux (kernel 4.4.0) but we've also run FreeBSD and Solaris and found similar results. Furthermore, our virtualization studies focus on KVM; however, we have also run VMware ESX and see similar results.

2.6.2 CPU Simulations and Analytical Models

To evaluate MIX TLBs, we need to go beyond existing hardware platforms. Unfortunately, current cycle-accurate simulators cannot run fast enough to collect meaningful data for all the long-running, big-data workloads (with multiple OSes, and hypervisors) we require for our CPU studies. Therefore, like most recent work on TLBs [84, 169, 29, 33, 123, 30, 37], we use a combination of tracing, performance counter measurements, functional cache hierarchy and TLB simulations, and anaytical modeling to estimate overall impact on program execution time. We use Pin [145] to collect memory traces. We extend Pintools with Linux' pagemap to include physical addresses with the virtual addresses that Pin normally generates. We select a Pinpoint region of 10 billion instructions and correlate traces with performance counter measurements to verfiy that the sampled region is representative of the workload.

These traces are passed to a functional simulator – used to assess TLB and cache hit rates – that models multi-level TLBs, hardware page table walkers, and a cache hierarchy. Our baseline is a split TLB hierarchy from Intel Haswell systems; further, we model area-equivalent hash-rehash and skewed TLBs, with prediction-based enhancements. Finally, we model an area-equivalent MIX TLB hierarchy.

We use the hit rates from our functional simulation to, like past work [84, 169, 29, 33, 123, 30, 37], feed into an analytical model that uses the performance counter data to weight the performance impact of TLB hits, misses, and cache accesses.

2.6.3 GPU Simulation

Our GPU studies use cycle-level CPU-GPU simulation based on gem5-gpu, running Linux, and modeling an x86 architecture. Like recent work [172, 173, 175], we model 128-entry, 4-way set-associative TLBs for 4KB pages per shader core. We also model split TLBs for 2MB pages (32-entry, 4-way) and 1GB pages (4-entry, fully-associative).

2.6.4 Workloads

Our CPU studies use two sets of applications. The first consists of all workloads from Spec and Parsec [40]. We scale the inputs of these workloads so that their total memory footprint is roughly 80GB. The second set uses big-memory workloads (*e.g.,* gups, graph processing, memcached, workloads from Cloudsuite [82]), also tuned to 80GB.

Our GPU studies, like recent studies [172, 173, 175] use workloads from Rodinia [59]. Ideally, our GPU studies should use the same big-memory sizes as our CPU studies, but this makes simulations infeasibly slow. We therefore scale our inputs to 24GB memory footprints.

2.7 Evaluation

We now present an evaluation of MIX TLB in two steps – first, a study of OS page allocation patterns and second, quantification of the benefits of MIX TLBs.

2.7.1 OS Page Allocation Characterization

Several factors affect the OS' page size distribution. For example, suppose we run Linux with transparent hugepage support (THS) [22]. As the program makes memory allocation requests (*e.g.,* malloc(), or mmap()), the OS earmarks virtual pages, using its virtual memory area data structure [64]. If these requests are to large amounts of memory, several contiguous virtual pages are reserved. Virtual pages are lazily allocated physical pages, as the program page faults through the virtual pages. The OS consults its free pool of physical pages for this. THS tries to defragment memory sufficiently to maintain swathes of contiguous free physical pages. If there is enough free physical memory for superpages, THS can assign 2MB physical pages to 2MB regions of virtual addresses, generating superpages. Further, if the program page faults through the virtual pages in ascending order, they are handed contiguous physical pages.

Instead of THS, Linux can also use libhugetlbfs. This is a special library that administrators or programmers have to explicitly link to [123]. Users can specify a superpage preference (*e.g.,* 2MB or 1GB). At link time, programs reserve a pool of memory. Superpages are allocated from this pool; when this pool is used up, small pages are allocated from other memory locations. Like THS, libhugetlbfs relies on lazy physical memory allocation and OS degfragmentation of physical memory.

Page size distributions: Figure 2.9 quantifies the page distributions we see on our real-system CPU and GPU experiments, both running Linux. The graphs show the fraction of total memory footprint backed by superpages (both 2MB and 1GB). As we have detailed, physical memory fragmentation impacts the frequency of superpage allocation. Therefore, we vary the level of memory fragmentation by running the microbenchmark memhog [169, 168], which allocates memory randomly across a fraction

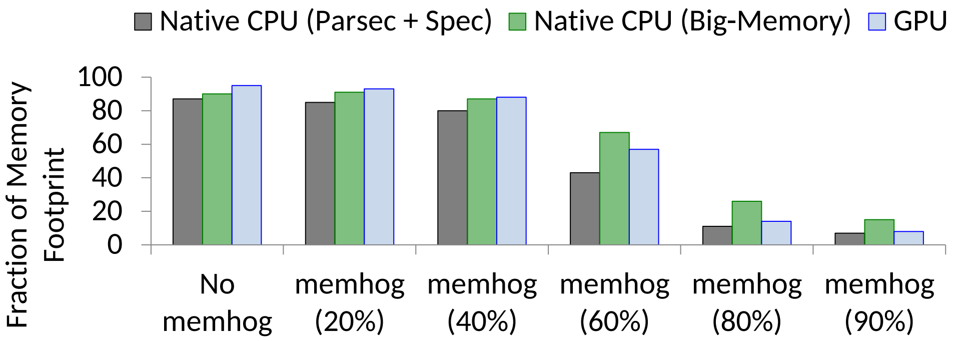


Figure 2.9: Fraction of memory footprint occupied by superpages, as fragmentation varies. Results shown for native CPUs and GPUs.

of system memory, in the background. For example, memhog (40%) on the x-axis indicates that memhog is fragmenting 40% of the 80GB system memory in the background. We show average numbers for classes of workloads (*e.g.,* Parsec + Spec, big-memory workloads) as per-workload numbers follow these trends. Figure 2.9 shows three regimes of page distributions.

Superpages dominate: With moderate amounts of memory fragmentation, superpages cover most of the application's memory needs. For example, even with memhog fragmenting 40% of physical memory, more than 80% of a CPU's or GPU's workload is covered with superpages, on average.

Neither small pages nor superpages dominate: When memory fragmentation further increases, the memory footprint is more equally divided among small pages and superpages. For example, memhog with 60% finds that 40-60% of CPU, and 55% of GPU footprints are backed by superpages.

Mostly small pages: When fragmentation becomes severe, the bulk of the memory footprint is backed with small pages.

Figure 2.10 shows that similar trends hold for virtualized workloads. To create memory fragmentation and system load, we first consolidate as many VMs on the same machine as possible. Each consolidated VM is provided 10GB of memory; therefore, 8 of them use up all 80GB of available physical memory. In addition, we run memhog

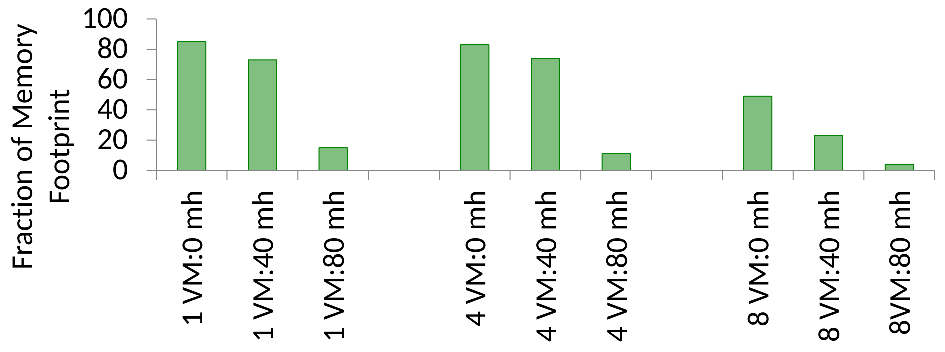


Figure 2.10: Fraction of memory footprint occupied by superpages, as a function of the memory fragmentation and VM consolidation. Results are for virtualized CPU workloads. N VM: M mh stands for N consolidated VMs, each with memhog running at M%.

within each VM, fragmenting a percentage of each VM's footprint. We expect that higher VM consolidation and more aggressive memhog use will reduce the frequency of superpages.

Figure 2.10 shows that OSes running in VMs can counter non-trivial amounts of memory fragmentation, producing lots of superpages. For example, even 4VMs with memhog of 40% each, see more than 70% of memory is allocated in superpages. Naturally, as system load increases, small pages dominate. For example, like recent work [170, 171, 94], we find that as more VMs are in the system and memory pressure increases, optimizations like page sharing [94] and NUMA migrations [88] preclude heavy use of superpages.

Overall, this data suggests many system factors influence page size distributions, and therefore, systems experience a variety of such distributions. It is therefore vital to implement efficient TLB support for mixed page sizes.

Contiguous superpages characterization: MIX TLBs rely on contiguity among superpages when they are present. Figure 2.11 quantifies the amount of superpage contiguity for the workloads and configurations from Figures 2.9 and 2.10 where at least one superpage is present. Figure 2.11 quantifies average contiguity per workload (numbered in ascending order of superpage contiguity on the x-axis). Average contiguity is measured as follows. We scan the entire page table and identify runs of contiguous

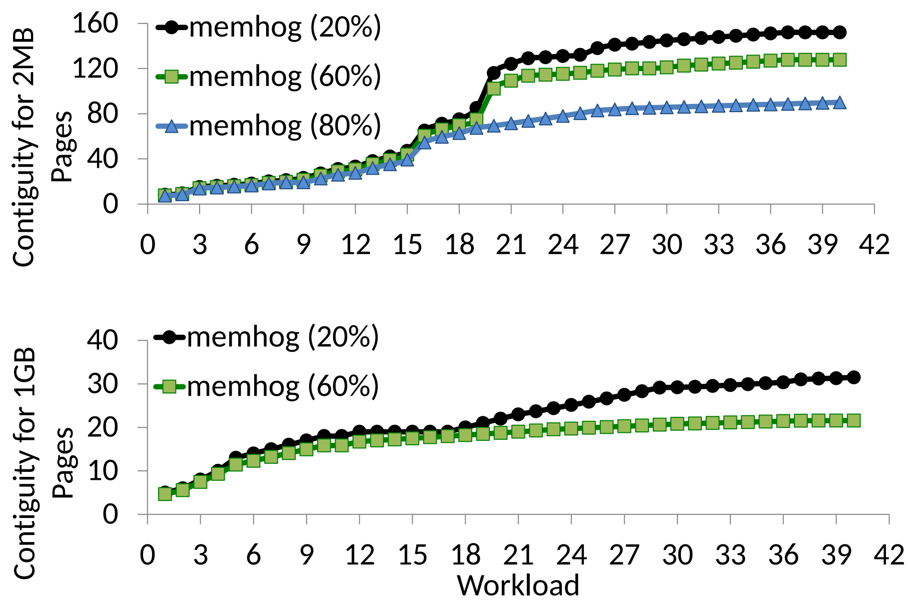


Figure 2.11: Average superpage contiguity for native and virtualized CPU, and GPU workloads. We show trends as memory fragmentation is increased with memhog, separately for 2MB and 1GB superpages.

superpages. We divide this contiguity by the number of translations. For example, suppose we have a page table with 4 entries, where the first 2 translations are singletons, but the last two are contiguous. We calculate that the average contiguity is $(1+1+2\times2)/4$. We separate results for 2MB superpages and 1GB superpages, varying the amount of memory fragmentation using memhog.

Figure 2.11 shows that superpages themselves – and not just their constituent 4KB page regions – are usually allocated contiguously in virtual and physical addresses. For example, consider memhog fragmenting 20% of physical memory. Figure 2.11 shows that most benchmarks have average 2MB page contiguity greater than 80. This means that 80+ 2MB superpages can potentially be coalesced in our TLBs. Since CPUs (*e.g.,* see Intel's Sandybridge, Haswell, and Skylake TLBs) use 16-set L1 TLBs, this is sufficient to entirely offset mirrors for L1 MIX TLBs. L2 TLBs usually have 64-128 sets; while memhog at 20% and 60% see enough 2MB page contiguity to offset this consistently, contiguity does drop with more fragmentation. Nevertheless, even in these cases, it is enough (80+) that it can sufficiently (though not entirely) enable coalescing to counter mirroring.

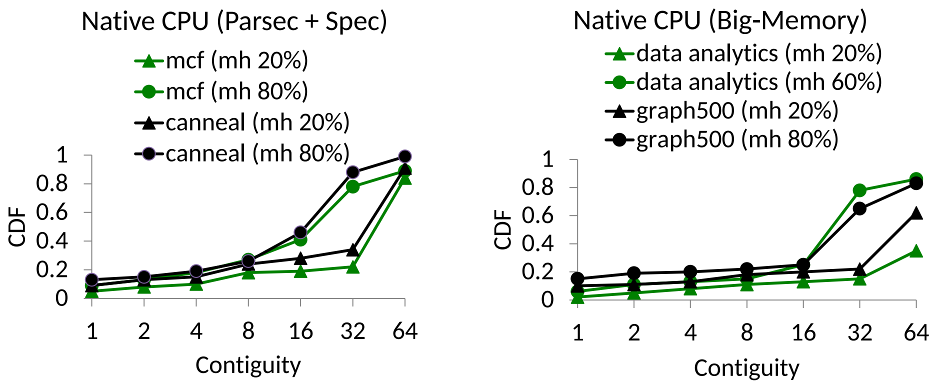


Figure 2.12: Superpage contiguity CDF as memhog varies, for native CPU workloads.

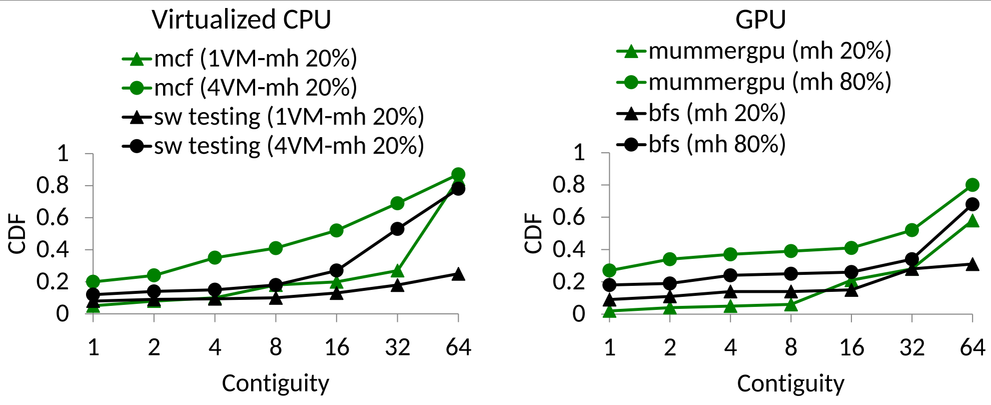


Figure 2.13: Superpage contiguity CDF as memhog varies, for virtualized CPU and GPU workloads

Figure 2.11 also shows contiguity for 1GB pages. Since 1GB pages require much larger defragmented physical memory regions than 2MB pages, they are harder to form. As a result, the number of contiguous 1GB pages is usually lower than 2MB pages. Most workloads see 20-30 contiguous 1GB pages, even with relatively high fragmentation when memhog is 60%. Fortunately, since this covers 20-30GB of memory in an 80GB memory system, this amount of contiguity is good enough for effective coalescing.

Figures 2.12 and 2.13 focus on superpage contiguity in terms of the cumulative distribution functions (CDFs) for native CPU, virtualized CPU, and GPU workloads. Once again, memory fragmentation is controlled using memhog, and virtualized results also rely on VM consolidation to generate load. Applications with higher contiguity see the largest increases in CDF values further along the x-axis. Figures 2.12 and 2.13 show that all these workloads see considerable contiguity, even when system fragmentation is high.

2.7.2 Results

We begin by comparing the performance of MIX TLBs against commercially available Intel Haswell TLB configurations. Note that while we refer to this as a split configuration, it uses split L1 TLBs, but partly-split L2 TLBs (*i.e.,* 4KB and 2MB translations are hashed-rehashed in 1 TLB, while 1GB translations are cached in a separate TLB). We then also compare MIX TLBs to simulated multi-indexing schemes (*i.e.,* hash-rehash and skew TLBs at both the L1 and L2 levels for all page sizes). Finally, we demonstrate how MIX TLBs perform in tandem with past work on COLT.

Comparisons to split TLBs: Figure 2.14 shows performance improvements using area-equivalent MIX TLBs versus a Haswell style TLB. To conserve space, we pick representative benchmarks from Spec + PARSEC, the big-memory workloads, and the GPU applications. We also show average results for the remaining workloads in each category.

We separate results for several cases. For native CPU workloads, we first use libhugetlbfs to try to use 4KB, 2MB, or 1GB pages exclusively. We then run native CPU workloads on a system with transparent hugepage support or THS enabled, where Linux attempts to allocate as many 2MB pages as possible, backing off to 4KB pages if this is not feasible. We also show results for virtualized CPU workloads, with 1 VM, and then a consolidated system with 4 VMs. The VMs are configured to support whatever mix of 4KB, 2MB, and 1GB pages the guest OS and hypervisor think are appropriate. Finally, we show GPU workloads on non-virtualized systems.

Figure 2.14 shows that MIX TLBs outperform commercial TLBs comprehensively, frequently in excess of 10%. For setups where small pages are prevalent (*e.g.,* 4KB bars), we see more than 8% performance improvements on native and virtualized CPUs, as well as GPUs. This is because split TLBs cannot use the 2MB/1GB L1 TLBs, and the 1GB L2 TLBs for 4KB pages, while MIX TLBs do not have this utilization problem.

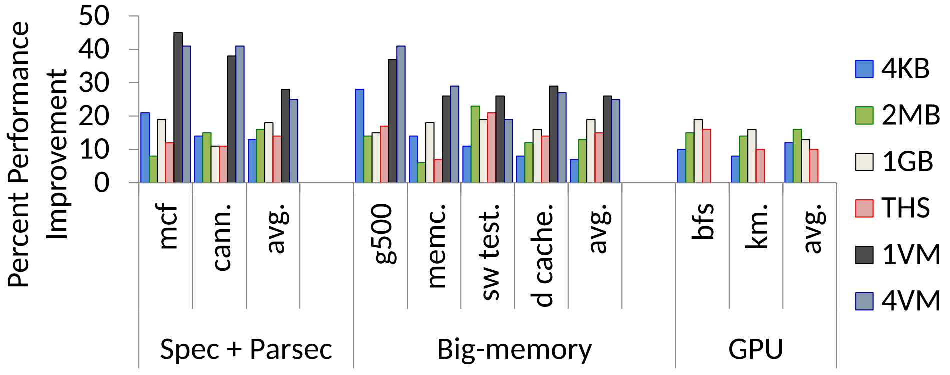


Figure 2.14: Percent performance improvement from MIX TLBs compared to area-equivalent split TLBs.

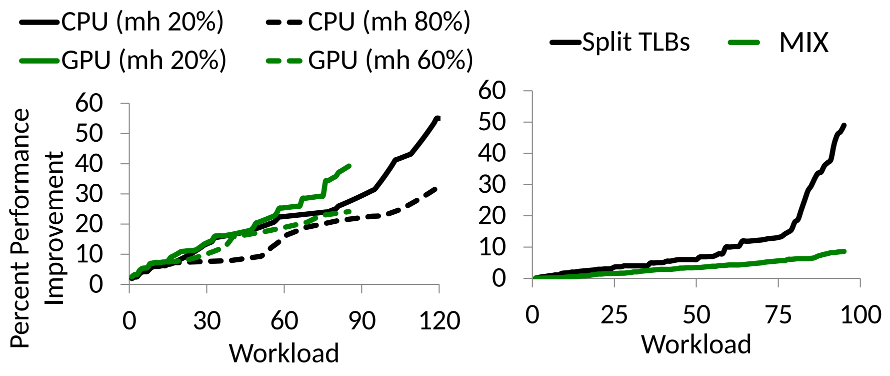


Figure 2.15: (Left) percentage performance improvement of MIX TLBs versus split TLBs, with memhog varying; (Right) Performance overheads of split TLBs and MIX TLBs compared to ideal hypothetical TLBs which never miss.

Figure 2.14 also shows that MIX TLBs perform well when superpages become more prevalent. For example, in the 2MB or THS cases, where 2MB pages become more common, MIX TLBs achieve better performance than split because they can utilize all hardware for 2MB pages, not just 2MB page TLBs. And these gains are even higher, in excess of 12%, for 1GB pages, which can only use small 1GB page TLBs in the split.

Unsurprisingly, MIX TLBs are particularly useful when TLB misses become more expensive. Therefore, for virtualized workloads, where TLB misses necessitate expensive two-dimensional page table walks [170, 32, 85], 40%+ performance improvements are seen. Similarly, GPUs, which experience heavy TLB miss traffic [172, 173, 175], enjoy significant performance benefits for any distribution of page sizes.

Figure 2.15 sheds further light on performance benefits, in the presence of memory fragmentation. The graph on the left shows MIX TLB performance improvements over split TLBs, as memhog fragments 20% and 80% of CPU memory, and 20% and 60% GPU memory. We arrange the workloads (numbered on the x-axis) in ascending order of performance benefits. As expected, increasing memory fragmentation does reduce performance as it reduces the incidence of superpages; nevertheless, MIX TLBs consistently outperform split TLBs by 20%+.

The graph on the right of Figure 2.15 compares how well MIX TLBs do versus a hypothetical ideal TLB which never misses and cannot hence be realized. We plot curves for the performance overheads experienced by split TLBs and MIX TLBs versus this ideal TLB. The lower the y-axis values, the better. While almost a third of the split Haswell TLBs experience a 10%+ performance deviation from the ideal scenario, MIX TLBs always achieve under 10%.

Comparisons to multi-indexing methods: We now compare performance and energy benefits of MIX TLBs versus area-equivalent skew-associative and hash-rehash (enhanced with the best prediction strategies [161]) approaches. Figure 2.16 shows these approaches for native and virtualized CPUs, as well as GPUs. We plot each workload along two dimensions. On the x-axis, we plot percent performance improvement versus the split TLB design. On the y-axis, we plot the percent address translation energy saved, also versus split TLBs. Therefore, we desire points at the top right quadrant of this space. The graph on the left shows skew-associative TLBs (blue) and hash-rehash TLBs (green), while the graph on the right shows MIX TLBs.

Figure 2.16 shows that MIX TLBs have better performance *and* energy than state-of-art multi-indexing schemes. Even the presence of operations like mirroring, which do increase energy by filling into multiple TLBs sets, are dwarfed by the big energy savings from decreasing TLB misses and hence cache/memory references. Another big source of energy savings comes from the relative simplicity of MIX TLB implementations; for

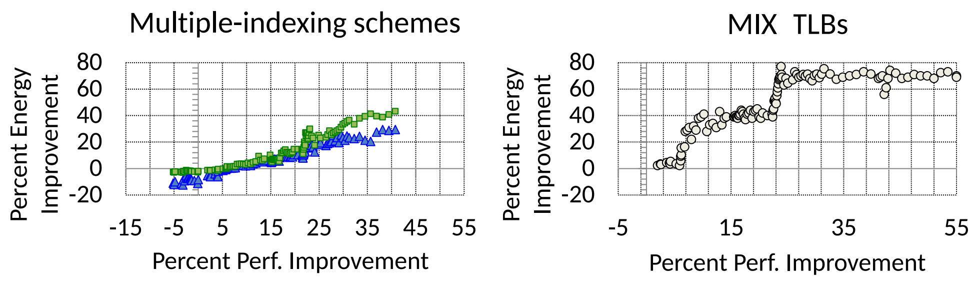


Figure 2.16: (Left) performance-energy tradeoffs for skew-associative TLBs with prediction (blue) and hash-rehash with prediction (green); and (Right) MIX TLB performance-energy tradeoffs.

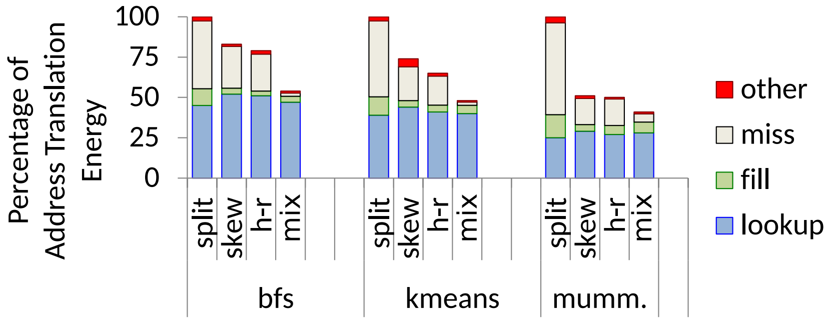


Figure 2.17: Percentage of address translation dynamic energy devoted to various TLB maintenance operations.

example, we find that skew-associative TLBs suffer area overheads from requiring time-stamp counters for good replacement policies [186]. Therefore, area-equivalent skew-associative TLBs have fewer entries than MIX TLBs. Hash-rehashing is indeed more energy efficient than skew-associativity but still needs to access a predictor structure, hurting its energy compared to MIX TLBs.

Note also that multi-indexing schemes can degrade performance and energy. This occurs when TLB hits are frequent but have to go through more complex multi-step lookups when the page size predictor makes mistakes. MIX TLBs do not suffer from these problems.

Dynamic energy breakdown: MIX TLBs achieve energy efficiency from their shorter runtime, which reduces leakage energy. It is more challenging, however, to identify the sources of savings in dynamic energy because MIX TLBs do have some more sophisticated operations (*e.g.,* mirroring). Figure 2.17 therefore quantifies the contribution of

TLB energy devoted to lookups, page table walks, TLB fills after the miss, and other operations like TLB invalidations. We focus on GPU TLB results but the trends remain the same in other applications too. The y-axis is normalized to the total energy expended on Haswell split TLBs.

Figure 2.17 shows that most energy is used for lookups and misses. This is because all loads and stores result in lookups, while misses are expensive, invoking multiple memory references through the memory hierarchy. In contrast, the energy on TLB fill is much lower. Therefore, mirroring, which occurs only on fills, does not affect overall energy substantially. Note also that unlike multi-indexing approaches, which increase lookup energy due to complex accesses with predictors, MIX TLBs leave lookup energy largely unchanged.

Scaling TLBs: We now focus on studying how TLB scaling, specifically with the number of sets, impacts MIX TLBs. Naturally, with more sets, we need more superpage contiguity to coalesce sufficiently to offset mirroring. Therefore, beyond the 64-128 set count maintained by Sandybridge and Haswell systems, we have also studied hypothetical TLBs with 512 sets. In general, we find that even though many workloads do not exhibit sufficient superpage contiguity to completely offset 512 mirrors, they still achieve 80+ pages of contiguity. This is usually enough for good performance. We have found that 512-set TLBs achieve within 13% of the performance of ideal TLBs which never miss.

Complementing COLT: Finally, MIX TLBs are orthogonal to past work on coalesced TLBs or COLT [169]. The original COLT work proposed coalescing contiguous small page translations into single TLB entries. However, an extension, which we call COLT++, may also coalesce contiguous superpages in split TLBs. Each of the split TLBs independently performs coalescing on their respective page size translations. We quantify the benefits of these approaches in Figure 2.18, comparing them to two other data points. The first is an area-equivalent MIX TLB. The second combines COLT with MIX TLBs. In this approach, we design a single set-associative TLB that can support multiple concurrent page sizes; however, there we can also coalesce contiguous small

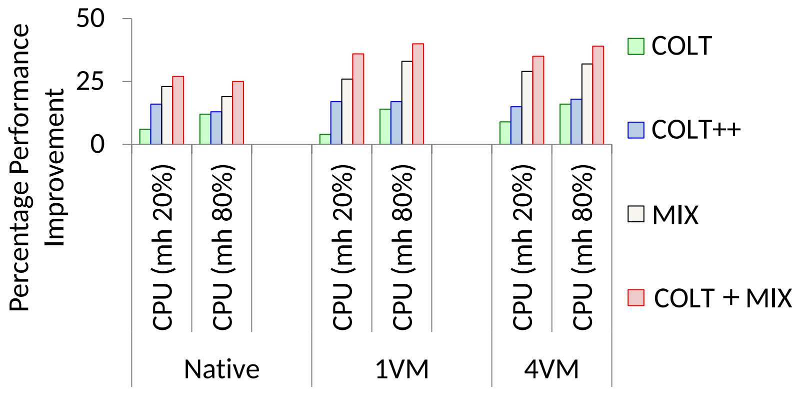


Figure 2.18: Compared to split TLBs, performance improvements from MIX TLBs and their combination with COLT.

pages. To compare fairly against past work [169], we assume that we can coalesce up to 4 contiguous small pages.

Figure 2.18 shows the average performance improvements of these various approaches versus Haswell-style split TLBs. We compare native and virtualized workloads, varying fragmentation with memhog. COLT can be helpful, but mostly when small pages dominate. In the presence of superpages, they cannot provide benefits. This explains the relatively low performance benefits when fragmentation is low (memhog 20%). COLT++ helps when superpages are frequent. On average, there are 8-10% performance differences versus COLT. However, MIX TLBs outperform even these cases because they can utilize all the TLB hardware for any distribution of page sizes. Further, combining MIX TLBs with COLT provide the highest performance, exceeding 20% benefits in all cases.

2.8 Conclusion

This work was motivated by the fact that modern TLB hardware is rigid in capacity allocation, despite the elasticity of the OS which can allocate many page size distributions. Many system factors affect these distributions, such as workload characteristics, system fragmentation and uptime, etc. There is a glaring gap between the richness of memory allocation at the software level, and modern TLB hardware.

We show one way of correcting this problem, with MIX TLBs, an energy-efficient

TLB that uses all its resources to seamlessly adapt to any distribution of page sizes. We show its benefits for native CPUs, virtualized CPUs, and CPU-GPU systems. Further, we believe that its simple implementation makes MIX TLBs ready for quick adoption.

Chapter 3

Scalable Distributed Shared Last-Level TLBs

3.1 Introduction

The advent of "big data" workloads with ever-increasing memory needs continues to pose performance challenges for modern computer systems. One important challenge is the question of how to achieve efficient virtual-to-physical address translation. Efficient Translation Lookaside Buffers (TLBs) are central to achieving high-performance address translation as they help avoid expensive multi-level page table walks.

TLB performance depends on three attributes – access time, hit rate, and miss penalty. Recent studies improve TLB hit rates using hardware-only or hardware-software co-design techniques like sub-blocking [201], coalescing [66, 169, 163], clustering [168], part-of-memory optimizations [147, 184], superpages [170, 202, 158, 133], direct segments [29, 84], and range translations [123, 86]. Others have used prefetching and speculative techniques to support the illusion of higher effective TLB capacity [38, 39, 189, 122, 170, 35, 28]. Similarly, synergistic TLBs, which evict translations between per-core TLBs, can improve hit rates [197]. Shared last-level TLBs have also been proposed [37] to improve the overall hit rate by avoiding replication of shared translations that occur in multi-threaded programs or multi-programmed workloads using shared libraries and OS structures. Finally, some studies have reduced TLB miss penalties by optimizing MMU caches, which are used to accelerate TLB misses [33, 27].

Unfortunately, many of these approaches have side-stepped the attribute of TLB *access time*. Consider, for example, shared TLB organizations. Processor vendors implement two-level TLBs private to each core today. However, recent academic work has shown that replacing them with an equivalently-sized *shared* (among cores) L2 TLB eliminates as much as 70-90% of the page table walks on modern systems [37]. However, sharing also results in larger structures that are physically further from cores, resulting in longer access latency. Recall that address translation latency is on the critical path of *every* L1 cache access. Consequently, a TLB sub-system with more TLB hits may not be attractive if each hit actually becomes slower. As the virtual memory demand from applications continues to increase, scaling TLB size but keeping them fast is a key research challenge.

Our goal is to translate the hit rate benefits of shared TLBs to overall speedup. This requires a conceptual re-think of how we architect a scalable shared TLB hierarchy. The challenge with a multi-banked monolithic shared L2 TLB structure is that it suffers from high latency. A natural alternative is a distributed shared L2 TLB, akin to NUCA LLCs. Each distributed shared TLB slice can be made small to keep access latency low. Unfortunately, this makes TLB access non-uniform, depending on the location of the slice where the translation is cached. Our studies on a 32-core Haswell system show that a distributed shared L2 TLB consequently *degrades* performance by 7%, despite having 70% fewer misses on average than private L2 TLBs. This is because TLB accesses are more latency critical than data cache accesses.

We propose NOCSTAR (**NOC**s for **s**calable **T**LB **ar**chitectures), a design methodology to architect scalable low-latency shared last-level (SLL) TLBs. NOCSTAR relies on the latency characteristics of on-chip wires and the bandwidth characteristics of address translation requests to realize a lightweight specialized interconnect that provides near single-cycle access to remote shared TLB slices, however far they may be on-chip. Consequently, NOCSTAR provides the hit rate benefits of shared TLBs at the access latency of private TLBs via the following features:

① High capacity: NOCSTAR offers higher hit rates than private L2 TLBs by eliminating replication and improving utilization.

② Low lookup latency: NOCSTAR achieves low lookup latency by replacing a monolithic shared L2 TLB structure with smaller TLB slices distributed across cores.

③ Low network latency: NOCSTAR employs a light-weight interconnect to connect cores to the distributed TLB slices. This interconnect provides near single-cycle latencies from any source to any remote TLB, reducing network traversal latency.

The confluence of these features enables NOCSTAR to offer almost all (i.e., within 95%) of the performance of an ideal, zero-interconnect-latency shared TLB. With an area-equivalent configuration (that conservatively reduces TLB sizes to account for our interconnect area), NOCSTAR outperforms private L2 TLBs on 16-64 core Haswell systems by an average of 1.13× and up to 1.25× across a suite of real-world workloads.

3.2 Background and Motivation

We first study conventional private L2 TLBs, and compare to shared L2 TLB alternatives proposed in prior work [37]. As we scale the size of the shared TLB, a practical design would involve banking this monolithic structure. We evaluate this design and ultimately find that distributing the TLB slices across cores with a fast NOC is a better choice.

While NOCSTAR is applicable to both instruction and data TLBs, we focus on the latter. Our focus is driven in part by the fact that the data-side TLB pressure is growing with the prevalence of big-data workloads [29, 84, 86, 35, 66, 133].

Throughout this chapter, we use the term *TLB access latency* to refer to *TLB's SRAM lookup latency + interconnect latency.*

3.2.1 Limitations of Private TLBs and Promise of Shared TLBs

Private two-level TLBs are a staple in modern server-class chips like Intel's Skylake or AMD's Ryzen processors. For example, Intel's Skylake chip uses 64-entry L1 TLBs backed by 1536-entry, 12-way set associative L2 TLBs per core. Unfortunately, private L2 TLBs suffer from the classic pitfalls of private caching structures – i.e., replication and poor utilization [37]. Consider the problem of replication. Multi-threaded applications running on a multi-core naturally lead to replication of virtual-to-physical translations across private L2 TLBs as they are part of the same virtual address space. Perhaps more surprisingly, even multiprogrammed combinations of single-threaded programs exhibit replication as different processes can share libraries and OS structures [37].

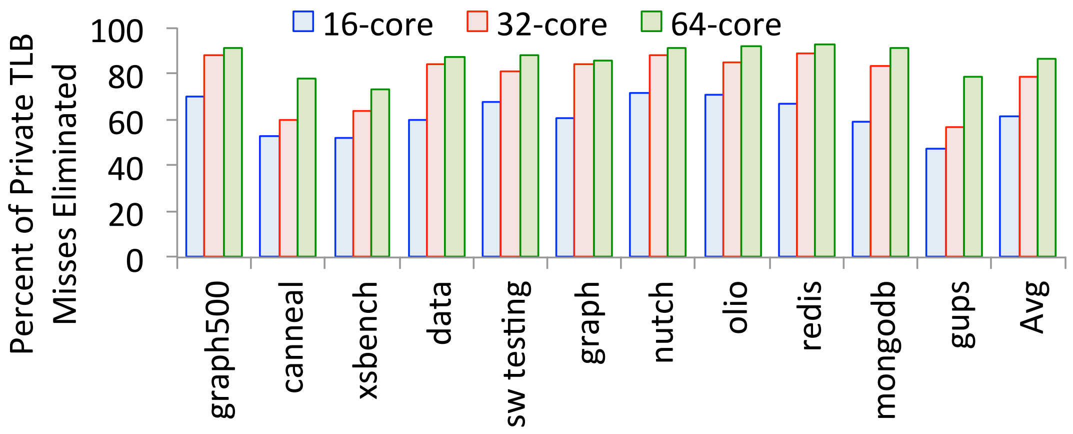


Figure 3.1: Percentage of private L2 TLB misses eliminated by replacing with a shared TLB. Results shown for 16-64-core systems.

Private L2 TLBs also suffer from poor utilization because chip-wide TLB resources are partitioned statically (usually equally) at design time. But this means that there are situations where, at runtime, a private L2 TLB on one core may thrash while its counterpart on another core may experience far less traffic [37].

Recent work has evaluated the potential of shared last-level TLBs (which we call *shared L2 TLBs*) [37]. Shared L2 TLBs eliminate the redundancy of private L2 TLBs and also seamlessly divide TLB resources to cores based on their runtime demands, overcoming the problem of poor utilization. Shared TLBs also offer implicit prefetching benefits; i.e., a thread on one core can demand (and hence prefetch) translations eventually required by threads on other cores. The original paper finds that shared TLBs eliminate as much as 70-90% of the misses suffered when using private L2 TLBs [37].

3.2.2 Shared L2 TLB Hit Rates

Figure 3.1 quantifies the benefits of shared L2 TLBs. Our system using Intel Haswell systems is described in Section 3.4. Figure 3.1 shows that shared L2 TLBs eliminate the majority of L2 TLB misses suffered by private TLBs. Note that for every one of our workloads, *the entire* private L2 TLB is used to store entries – that is, no translations are wasted. Furthermore, like prior work [66, 37], we found that absolute private L2 TLB miss rates range from 5-18% for our workloads. Naturally, the main reason these

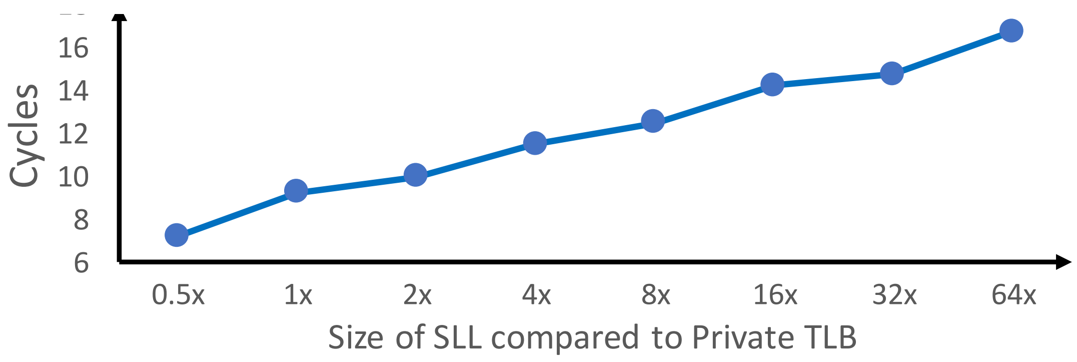


Figure 3.2: Access latency of SRAM TLB compared to number of entries in a TLB. Post-synthesis in 28nm TSMC PDK.

miss rates are harmful is the fact that each TLB miss is a particularly long-latency event.

Generally, the higher the core count, the more effectively the shared L2 TLB eliminates private L2 TLB misses. Consider, for example, a situation with 4 cores, and one with 16 cores. If private TLBs are N entries, the 4-core case can replace the private L2 TLBs with a shared L2 TLB with 4×N entries for the TLB resources. A 16-core case can realize a 16×N-entry L2 TLB instead. Naturally, we are therefore able to eliminate the replication and utilization problems of private TLBs even more effectively at higher core counts. Workloads with notably poor locality of access (e.g., `canneal`, `gups`, and `xsbench`) are particularly aided by shared TLBs at higher core counts.

3.2.3 Shared TLB Access Time

One might expect the hit rate improvements of Figure 3.1 to improve performance overall. However, TLB performance is influenced not just by hit rates, but also the following:

① **SRAM array lookup times:** L2 TLBs are typically implemented as SRAM arrays. Unfortunately, scaling SRAM arrays while ensuring fast access is challenging. We model SRAMs in TSMC 28nm technology node using memory compilers. Figure 3.2 quantifies access latency scaling as a function of the number of entries in the array (all numbers are post-synthesis). A 1536-entry L2 TLB (the size of private L2 TLBs in Intel Skylake) takes 9 cycles, while a 32×1536-entry design takes close to 15 cycles to access. Replacing

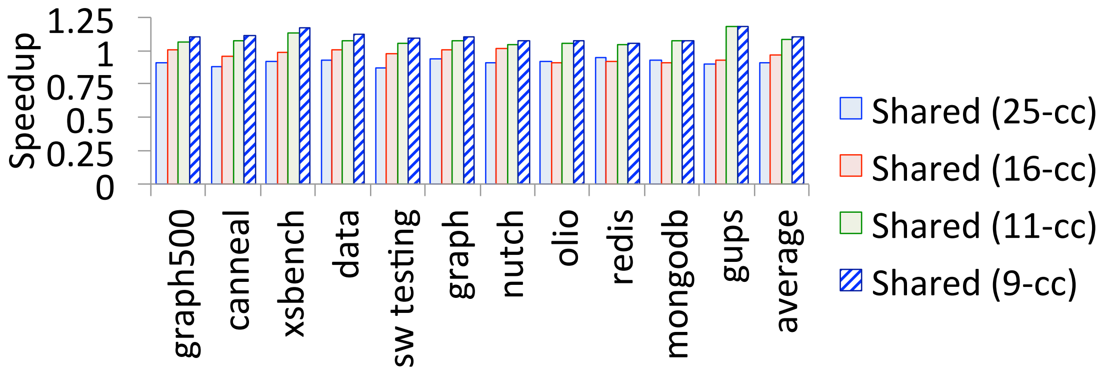


Figure 3.3: Speedups using shared multi-banked TLBs over private L2 TLBs. Shared TLB access latencies varies from 25 to 9 cycles.

private TLBs with an equivalently-sized shared TLB means that the shared structure grows from a 12K-entry structure for 8 cores (8×1536 entries) to a 96K-entry structure for 64 cores (64×1536 entries), increasing lookup times by factors of 2-4× Ultimately, this high access latency – which worsens as we need larger shared TLBs for higher core counts – counteracts the benefits of higher hit rates.

② **Interconnect traversal times:** The original paper on shared TLBs focused on monolithic designs where the entire structure was placed at one end of the chip [37]. Naturally, this design exacerbated access times further, due to additional interconnect delays to access the shared TLB location. This was observed to counteract the benefits in some cases even for a 4-core system [37]. Higher core counts further worsen this delay. For instance, for a 64-core system, the tiles at the top of the chip would require 8 hops in each direction to access the TLB. ③ **Bandwidth:** A key problem with the original shared TLB proposal is that accesses from multiple cores suffer from contention at the shared structure's access ports. This differs from the private L2 TLB scenario, where each core can access its private TLB without interference from other cores.

3.2.4 Shared TLB Performance

Figure 3.3 quantifies how attributes ①-③ counteract higher hit rates in determining the overall performance of shared monolithic L2 TLBs. We profile performance on a 32-core Haswell system using monolithic shared L2 TLBs versus private L2 TLBs. Based on our SRAM array memory compiler studies with 28nm TSMC, we determine that the private L2 TLBs have 9-cycle lookup times. These are consistent with other references that measure Haswell TLB lookup times and Intel's product manuals, which measure lookup latencies of 7-10 cycles for private L2 TLBs [111, 109]. For our shared L2 TLB, we vary the total access latency from 9 cycles (an unrealizable ideal case where the 32× larger SRAM array has access times that match the private L2 TLBs and the interconnect is zero-latency) to 25 cycles (a more reasonable estimate of the larger SRAM array plus interconnect latency). We bank the shared L2 TLB; we study designs with 16, 32, 64, and 128 banks. We plot results from the highest-performing banking configuration for each workload. Section 3.4 describes the rest of the system configuration. Note that all our experiments assume Linux 4.14 is running on the system, with support for transparent hugepages [169, 66]. In practice, we find that over half of the memory footprint of the workloads are implemented as superpages (see Section 3.5 for more details).

Figure 3.3 shows that despite better hit rates, the monolithic shared TLB can perform poorly. For example, at 25-cycle access latency, we see a 10-15% performance dip versus private L2 TLBs. Even worse, consider an unrealizable ideal network with zero interconnect latency (i.e., the only latency arises from port contention and SRAM array latency), which corresponds to the scenario where the shared L2 TLB access takes 16 cycles. Even here, the shared TLB shows little to no speedup over the private L2 TLB case.

3.2.5 Understanding Shared L2 TLB Access Patterns

We now study key aspects of shared TLB access patterns that can help us overcome access latency problems.

Shared L2 TLB contention across applications. Figure 3.4 captures information about contention at the shared L2 TLB. For every shared L2 TLB access, we plot the number of other cores with outstanding shared L2 TLB accesses. Figure 3.4 shows that more than 40% of the L2 TLB accesses occur in isolation; i.e., there is no other outstanding TLB access. Roughly another 20-30% of the L2 TLB accesses occur when there are only 2-4 outstanding shared L2 TLB lookups.

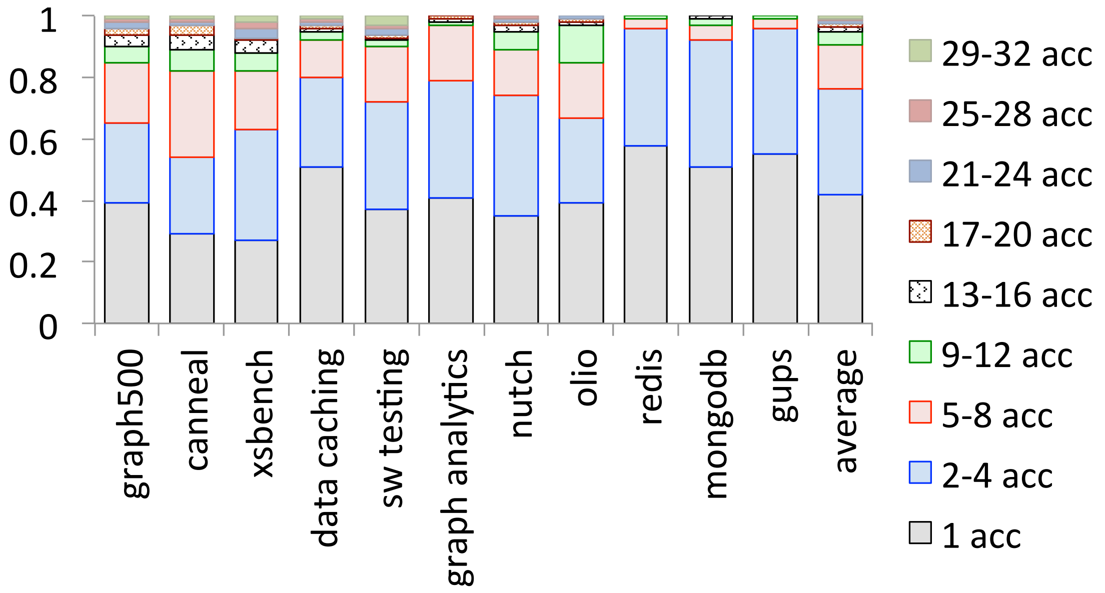


Figure 3.4: Fraction of L2 TLB accesses that occur concurrently with 1 other access, 2-4 other accesses, etc., on a 32-core Haswell system.

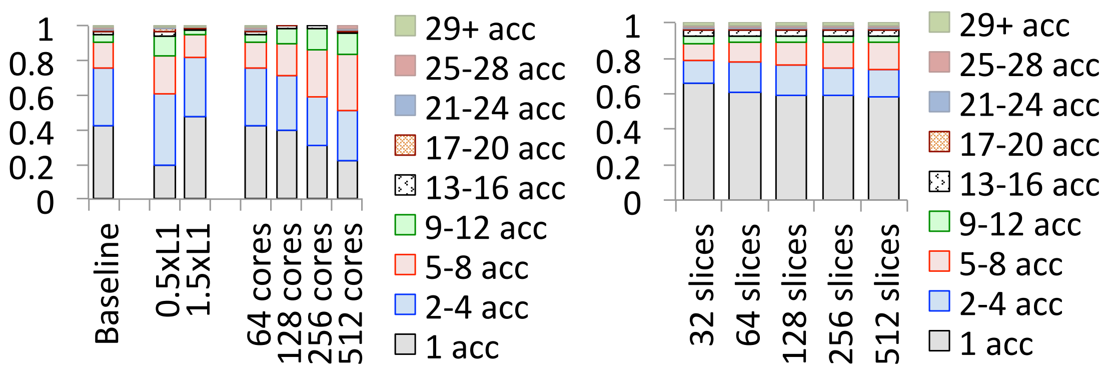


Figure 3.5: (Left) Fraction of L2 TLB accesses that occur concurrently with 1 other access, 2-4 other accesses, etc. Each bar averages results across workloads; (right) fraction of L2 TLB accesses to a TLB slice that occurs concurrently with 1 other access to that slice, 2-4 other accesses to that slice, etc. Each bar shows a distributed shared L2 TLB, where the number of TLB slices is equal to the number of cores.

Shared L2 TLB contention with varying L1 TLB size. The larger the L1 TLB, the fewer the shared L2 TLB accesses. Figure 3.5 (left) shows the impact of the L1 TLB size on shared L2 TLB contention. The `baseline` bar matches the average access distribution from Figure 3.4, while the `0.5×L1` and `1.5×` bars represent distributions as the private L1 TLBs per core are halved or increased by 50%. As one might expect, smaller L1 TLBs lead to more shared L2 TLB lookups. Consequently, the `2-4 access` and `5-8 access` portions of the bars increase significantly, implying greater contention. More interesting however are the trends towards bigger L1 TLBs as this reflects the direction processor vendors are going in. When we increase the L1 TLB sizes by 50%, we see contention dropping, with the `1 access` case dominating and taking up roughly 50% of the shared L2 TLB accesses.

Shared L2 TLB contention with varying core counts. Finally, Figure 3.5 also shows the impact of core count on shared TLB contention. The `baseline` represents 32-core Haswell; 0.5×L1 and 1.5× are for 32-core Haswell with half and 1.5 times the baseline L1 TLB size. The `64-512 core` results assume 64- to 512-core Haswell systems and we expect shared L2 TLB contention to increase with a higher number of core counts. However, not only does contention not increase at 64 cores, it only marginally increases at 128 cores (i.e., the `5-8 accesses` and `9-12 accesses` contributions increase by roughly 10% and 5% respectively). Only when we begin to approach 256 cores and beyond does contention visibly increase. However, we have also performed experiments where we have replaced the monolithic (banked) shared L2 TLB with a distributed shared L2 TLB, where the number of L2 TLB slices equals the core count. The graph on the right in Figure 3.5 showcases our results, this time quantifying the contention on average per TLB slice. As shown, even with high core counts (256-512 cores), roughly 60% of accesses to a single L2 TLB slice suffer no contention with concurrent accesses.

Takeways. The key takeaway from the three experiments above is the following – L2 TLBs must be accessed fast for performance but *concurrent accesses are rare.* This is true not just for system configurations today, but would continue to remain true and in fact drop further in future systems with larger L1s or more cores. Later in Section 3.5, we also validate this observation for a TLB miss "storm" microbenchmark (where we deliberately create high L1 TLB miss situations). This conceptual underpinning motivates our work - we design a specialized interconnect optimized for low latency rather than high bandwidth (required for concurrent accesses) to accelerate shared L2 TLB access.

3.2.6 Low-Latency Interconnects

On-chip wire delay. As technology scales, transistors become faster, but wires do not [100], making wires slower every generation relative to logic. This fact prompted research into NUCA caches [96, 126]. However, since clock scaling has also plateaued, wire delay in *cycles* remains fairly constant across generations. Long on-chip wires have repeaters at regular intervals, and take about 75-100 ps/mm [100, 60, 61]. Thus **it is possible to perform a 1-cycle traversal across the chip in modern technology nodes**, as recent chips have demonstrated [60, 61].

NOC traversal delay. The network latency (T) of a message in modern NoCs is denoted as [115]:

$$T = H \times (t_r + t_w) + \sum_{h=1}^{H} t_c(h) + T_s$$

H is the number of hops required to reach the destination, t_r is the router delay, t_w is the wire delay, $t_c(h)$ captures the contention at each router, and T_s is the serialization delay incurred when sending a wide packet over narrow links. The latency is directly directly proportional to H.

Challenges with designing low-latency NOCs. It is usually hard to build NOCs optimized for latency, bandwidth, area and power (see Table 3.1). Buses do not scale and each traversal is a broadcast. Meshes are the most popular due to their simplicity and scalability, as they rely on a grid of short links with simple routers (with low t_r) at cross-points. However, the average hop count H (and therefore latency) is much higher. High-radix NOC topologies (such as FBFly [127]) add long-distance links between distant routers, reducing H However, these naturally add more links (i.e., bandwidth), leading to extremely high area and power penalties due to multi-ported routers and crossbars. If we use a narrower datapath (i.e., reduced bandwidth), we can reduce area

Table 3.1: TLB interconnect design choices.

NOC	Latency	Bandwidth	Area	Power
Bus	✓	✗	✓	✗
Mesh	✗	✓	✗	✗
FBFly-wide [127]	✓	✓✓	✗✗	✗✗
FBFly-narrow	✗	✓	✗	✗
SMART [132]	✓	✓	✗	✗
NOCSTAR	✓	✓	✓	✓

and power to that of a mesh, but serialization delay T_s leads to higher latencies. Optimizations such as SMART [132] fall in between these extremes by enabling packets to dynamically construct bypass paths over a mesh, reducing the effective H. However, the paths are not guaranteed, and require expensive control circuitry to setup and arbitrate for, leading to false positives and negatives [132]. Moreover, buffers at routers in a Mesh, FBFly and SMART add high area and power overheads. NOCSTAR proposes an interconnect with $t_r = 0$, H=1, and $t_w = 1$, as we describe in the next section.

3.3 NOCSTAR Design

Our approach, NOCSTAR, organizes the SLL TLB as a distributed array of TLB slices (to reduce lookup latency) connected by a configurable single-cycle network fabric (to reduce interconnect latency).

3.3.1 TLB Organization: Distributed TLB slices

The overall organization of NOCSTAR is a logically shared last level TLB distributed across the tiles of a many-core system. It mirrors the design of NUCA LLCs [96]. Each slice is the same size or smaller than the size of current private L2 TLBs, thereby meeting the same area and power budgets.

- *TLB Entries:* Each entry in a slice includes a valid bit, the translation and a context ID associated with the translation.
- *Indexing:* Although optimized indexing mechanisms can be adopted for better performance, we use a simple indexing mechanism using bits from virtual address.

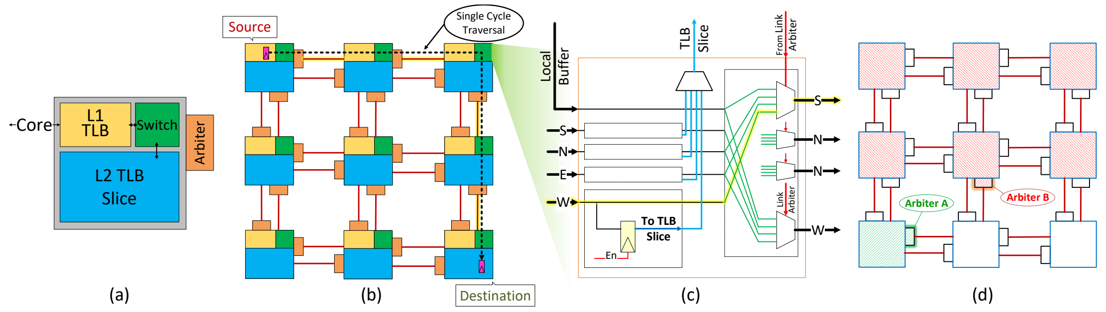


Figure 3.6: (a) TLB hierarchy near each core in NOCSTAR. (b) Source and destination of a request and the path of taken by the request. (c) Micro-architecture of the switch which enables single cycle traversal through the network. (d) Cores that can send requests to a given arbiter.

3.3.2 TLB Interconnect

We develop a dedicated side band NOC for communicating between the L1 TLBs and L2 TLB slices. As discussed in Section 3.2.6 and Table 3.1, directly adopting NOCs used between data caches today may not be the optimal design choice for a TLB interconnect. Instead, we develop a latchless, circuit-switched interconnect that can provide single-cycle connectivity between arbitrary source-destination pairs.

Datapath: Latchless Switches

The datapath in NOCSTAR leverages the fact that wires are able to transmit signals over 10+ mm within a GHz (Section 3.2.6). To enable single cycle traversal of packets in NOCSTAR we add a simple *latchless switch* next to each L2 TLB slice as shown in Figure 3.6(a). The switch is simply a collection of muxes as Figure 3.6(c) shows. The muxes are pre-set before a message arrives, as we will describe in Section 3.3.2. Figure 3.6(b) shows a request arriving from the *West* direction traversing the switch and directly getting routed out of the *South* direction, as selected by the multiplexers, without getting latched. A message gets latched only at the destination switch where it needs to be ejected out to the target L1 TLB or L2 TLB slice. For example, an L1 TLB at the top left corner can send a request within one cycle to the L2 TLB slice at the bottom right, as Figure 3.6(b) highlights. Each mux acts a like a *repeater*, and the entire traversal is similar to that of a conventional repeated wire [60, 61].

Bandwidth: This datapath is naturally lower bandwidth than a Mesh or FBFly as it does not have any buffers internally within the NOC. Moreover, unlike a FBFly which has more links, it cannot support multiple simultaneous transmissions unless they are using completely separate set of links. However, as we demonstrated earlier in Section 3.2.5, L1 TLB misses are infrequent – there is only one access 60% of the time, and 1-4 accesses 80% of the time, making this low-bandwidth NOC sufficient for our purpose.

Scalability: Each traversal over this network takes a single-cycle. For large chips running at very high frequencies, this might be multiple cycles by adding pipeline latches as we discuss in Section 3.3.2.

Control Path: Fine-Grained Circuit-Switching

We now describe the various steps involved in sending the messages.

Path Setup: For each traversal through the interconnect, all data links in the path have to be *acquired* before sending any kind of message. To ensure that the packet reaches the destination in a single cycle, *all* links in the path must be acquired in the same cycle. This is done using separate control wires. Each data link has an associated arbiter which can allocate the link to one of the requesting cores. Figure 3.7 shows an example of a core sending *requests* to all link arbiters in its path and receiving *grants* from each link arbiter before traversing the path. If any requester fails to acquire *all the links* in the desired path, because of any contention, it will wait and try again in the next cycle. This ensures that there are no packets traversing partial paths and thus avoids complexity. Once a path is *acquired*, the message can traverse through the datapath as shown in Figure 3.6(b).

Fanout from Switch: Each core has must have a way to setup a path to any of the slice present in the system. The width of the control wires for each arbiter depends on the routing policy adopted by the TLB system at design time. Consider an XY based policy in a system as shown in Figure 3.6(d). Each core is connected to the arbiter associated with a link through which the core can send a request. Thus, the number of wires going out of each core is $(num\_cores\_each\_row - 1) + ((num\_rows -$

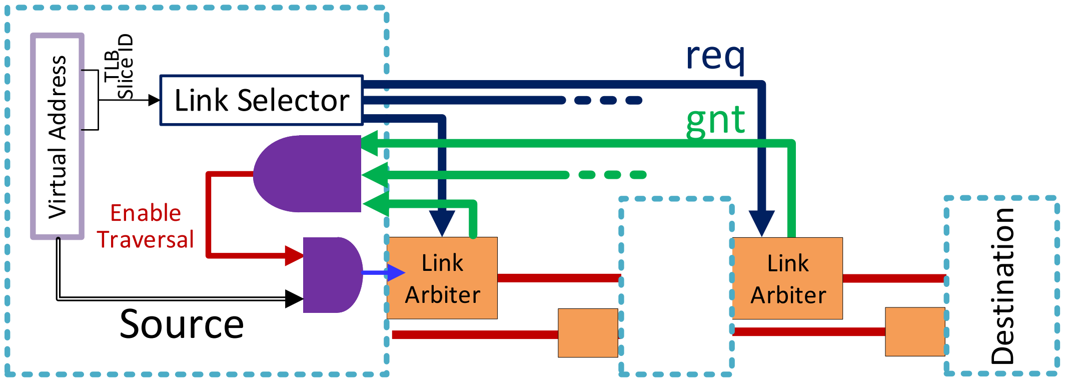


Figure 3.7: For setting up the path a core sends requests to all link arbiters in the path and waits for grants from them.

1) $\times$ ($num\_columns$)).

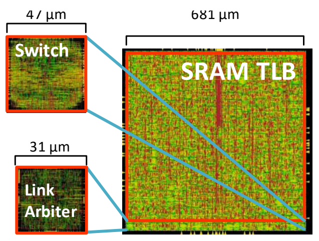
Link Arbiters: Each network link has an associated arbiter residing near the switch. The arbiter gets requests from any core which can send a TLB request/response packet through the link. This arbiter then selects one of the requesting cores and grants the link to it for the next cycle by setting the output mux to receive from the appropriate input port, and sending a 1-bit grant back to the requester, as shown in Figure 3.7.

Fanin at Link Arbiters: Depending on its physical location on-chip and the routing policy, different arbiters will have different number of requests coming in. For example, suppose we only allow XY routing. Figure 3.6(d) shows that the green Arbiter A for an X link can only have one requester, while the red Arbiter B for a Y link has six possible requesters.

Arbitration Priority: As the arbitration for each link is decentralized, there could be a possibility of livelock if two or more requests only acquire a partial set of links during each arbitration. To avoid this, the arbiters follow a static priority order among the requesters, to allot the links. In other words, a requester with higher priority will be guaranteed to get all its requested links. Further to avoid starvation, the static priority changes in a round-robin fashion every 1000 cycles.

Implementation

We implemented the NOCSTAR interconnect in TSMC 28nm with a 2GHz clock. Figure 3.8 shows the place-and-routed design. We observe the following.



Per Core	Power (mW)	Area (mm^2)
Switch	0.43	0.0022
4x Arbiters	2.39	0.0038
SRAM TLB	10.91	0.4646

Figure 3.8: Place-and-routed NOCSTAR tile in 28nm TSMC with the L2 TLB SRAM, switch and link arbiters highlighted and power/area of a switch and link arbiters for each slice in comparison to a SRAM based TLB slice. Target Clock Period = 0.5ns.

Critical Path. There are two sets of critical paths in the interconnect. On the *datapath*, a multi-hop traversal through all the intermediate switches needs to be performed within one clock cycle. Recall that the TLB interconnect is created at design-time. If timing is not met at the desired clock frequency, pipelined latches can be added at the maximum hops per cycle (HPC$_{max}$) [132] boundaries. This will increase the network traversal delay, but does not affect the operation of the design. Moreover, as core counts increase and tiles become smaller, the maximum hops per cycle will actually go up. On the *control path*, the critical path consists of the path setup request to the furthest link arbiter, link arbitration, and the grant traversal back to the core (Figure 3.7). We observed that the place-and-route tool placed all the arbiters close to the center of the design to reduce the average wire lengths to meet timing.

Area and Power. Figure 3.8 shows the post-synthesis power and area consumed by the NOCSTAR switch and arbiter. We contrast it with the cost of the L2 TLB SRAM present in the same tile. The area consumed by switch and arbiter is less than 1% of the tile's L2 TLB SRAM. The link arbiters, due to high fanin and tight timing, are the most power hungry component and key overhead. We can reduce this overhead by restricting the routing algorithm (and correspondingly the fanin), as discussed earlier in Section 3.3.2.

3.3.3 Timeline of L2 TLB Access in Nocstar

Figure 3.9 presents a timeline of address translation when there is an L1 TLB miss.

L1 TLB Miss. The L1 TLB miss triggers a circuit-switched path setup. The path setup can be performed speculatively during the L1 TLB access as well.

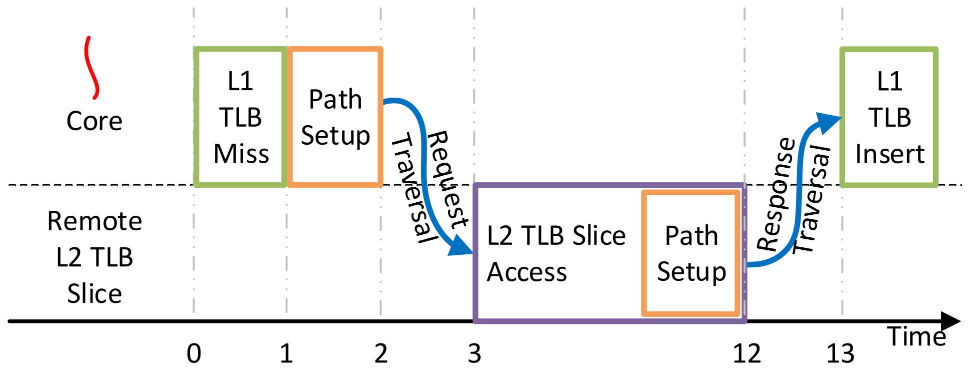


Figure 3.9: Timeline of a virtual address translation in case of an L1 TLB miss and remote L2 TLB access in Nocstar.

Request Path Setup. The remote TLB slice to which the translation is mapped is identified by the indexing. A path setup request is then sent to the arbiters of the links in the path. The grants from all the requests are ANDed to determine if the full path was granted or not. If not, the path setup is retried. If the full path is granted, the request is sent out.

Request Traversal. The TLB request is forwarded to the switch connected to the TLB slice (Figure 3.6(a)). No header or routing information needs to be appended, since the path is already setup. The request takes a single-cycle through all the intermediate switches, and is latched at the remote TLB slice and enqueued into its request queue.

L2 TLB Slice Access. The remote TLB slice receives the request and services the request. The translation may either exist or not. If it is a TLB hit, a response should be sent. The response contains the physical page associated with the virtual address in the request. A TLB miss would lead to a page walk which is discussed in Section 3.3.6.

Response Path Setup. A circuit-switched path for the response is requested. The response path can be setup speculatively, during the L2 TLB lookup, as a response will be sent to the requester regardless of access result.

Response Traversal. The response traverses the TLB interconnect within a single-cycle.

L1 TLB Insert. The requested translation is inserted into the requesting L1 TLB if it was a hit.

3.3.4 L2 TLB Access Latency and Energy

We quantify the benefits in latency and energy that NOCSTAR provides over a monolithic and distributed shared TLB.

Figure 3.10(a) shows the latency of a message when traversing different number of hops through the TLB interconnect in the different shared last-level TLB designs. We consider two cases.

Case 1: The requested translation is indexed in the slice of the requesting core: The virtual address is used to index into the SLL slice in the local node and the translation is returned to L1 TLB. The total latency incurred is equal to *lookup_latency* of the TLB slice for both Distributed and NOCSTAR designs. This is identical to private last-level TLB latency.

Case 2: The requested translation indexes to a remote slice: The required translation request is sent to the remote node containing the slice through a dedicated network. Once it reaches the destination node, the virtual address is used to index into the SLL slice and the translation is then sent back to the requesting slice. Upon receiving the translation response, the requesting core can then forward the translation to the L1 TLB. The total latency in this case is *lookup_latency* + *network_latency*. Here, NOCSTAR provides a latency advantage over both Monolithic and Distributed. Even when the maximum hops per cycle HPC_{max} in NOCSTAR goes down, it is still much faster than the distributed case.

Figure 3.10(b) shows the energy consumed by a message when traversing different number of hops through the TLB interconnect to understand trade-off spaces among the shared TLB designs. Most of the energy savings for the distributed design and NOCSTAR come from accessing a smaller SRAM structure than a monolithic(M) SLL TLB. Further, on the datapath, because of circuit switching, the energy consumed by an intermediate switch in NOCSTAR (N) is less compared to a switch in a traditional distributed network (D) with multi-cycle hops. However, NOCSTAR has a more expensive control path because of multiple request and grant wires spanning to all the link arbiters for simultaneous arbitration (Figure 3.7). For instance, to traverse 14 hops

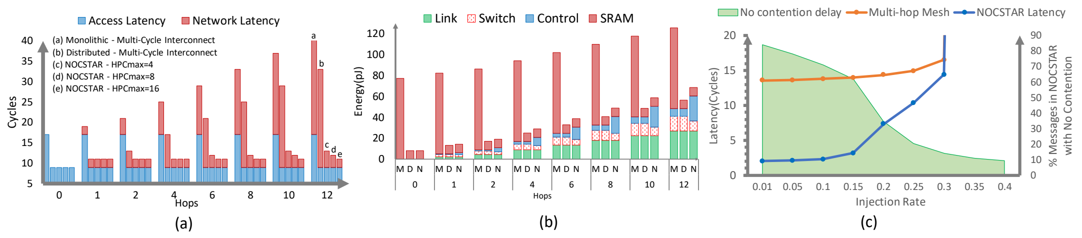


Figure 3.10: (a) Latency of each message in the TLB Interconnect in various configurations. (b) Energy consumed by each message in the TLB Interconnect in various configurations. (M)onolithic, (D)istributed, and (N)OCSTAR vs number of hops (c) Average latency of messages with respect to increasing injection rate in NOCSTAR interconnect compared to a multi-hop interconnect.

within a cycle, NOCSTAR will require 14 links to be arbitrated for simultaneously. This shows up as a slightly higher control cost than Distributed. However, the latency gains from this approach leads to an overall energy savings, as we discuss in Section 3.5.

3.3.5 Insertion/Replacement Policy

Like recent studies on TLB architecture, we assume that L1 and L2 TLBs use the lower-order bits of the virtual page number to choose the desired set using modulo-indexing, and use LRU replacement [169, 168, 170, 162, 35, 33, 39, 66]. Furthermore, like all recent work on two-level TLBs [66, 37, 169, 170, 168], we assume that the L1 and L2 TLBs are mostly-inclusive. Like multi-level caches, mostly-inclusive multi-level TLBs do not require back-invalidation messages [112].

3.3.6 Handling Page Table Walks

Suppose that a core suffers an L1 TLB miss and must look up the shared last-level L2 TLB. Suppose further that it determines that the TLB slice housing the desired translation lies on a remote node. If lookup of the remote node's TLB slice ultimately results in a miss, there are two options for performing the resulting page table walk. In the first option, the remote slice can send a *miss* message back to the requestor node, which must now perform the page table walk. In the second option, the remote node can itself perform the page table walk. Both approaches have pros and cons. Handling page table walks at the remote node is attractive in that it eliminates the need for a *miss* message to be relayed between the remote and requestor nodes. However, handling page table walks on the remote node also increases the potential for page table walker congestion; i.e., if multiple core's send requests to a particular remote slice and all of them miss, page table walks can be queued up.

3.3.7 TLB Shootdowns

A key design consideration involves how NOCSTAR responds to virtual memory operations performed by the OS. In particular, consider a situation where a page table entry is modified by the OS on a particular core. When this happens, the OS kernel usually launches inter-processor interrupts (IPIs) that pause other cores and run an interrupt handler that "shoots down" or invalidates the stale translation in the TLB. This operation requires care in NOCSTAR – specifically, it is now possible that multiple cores simultaneously relay a translation invalidation signals to a single TLB slice that houses the stale translation. This can quickly congest the system by cascading TLB invalidation lookups of a single TLB slice.

We sidestep this by designating some node(s) as the *invalidation leader(s).* In other words, even though *any* core can receive IPIs, and each core invalidates its private L1 TLB, only specific cores are permitted to then relay invalidation signals to the shared TLB. For example, if core 0 is considered the invalidation leader, *any core* that receives an IPI has to relay a message to core 0. Core 0 in turn relays a message to the relevant shared TLB slice to invalidate the stale translation. The actual TLB invalidation process for NOCSTAR from here on out mirrors that of a private L2 TLB. That is, during a private L2 TLB invalidation event, accesses to other translations in the private L2 TLB can be made; similarly, during the invalidation of a shared L2 translation, accesses to other translations (within the same slice or to other slices) are permitted. In Section 3.5, we study our approach. The ideal scenario is a middle ground where the number of leaders is far fewer than the core count, but where it is not so small that the messages become congested at any particular leader core.

3.4 Methodology

Simulation framework: We evaluate the benefits of NOCSTAR using an in-house cycle-accurate simulator based on Simics [16]. We model Intel Haswell systems [109] running Ubuntu Linux 4.14 with transparent superpages (which is the standard configuration). We model Intel Haswell cores with 32KB 8-way L1 instruction/data caches with 4 cycle access times, 256KB 8-way L2 caches with 12 cycle access times, and an LLC with 8MB per core and 50 cycle access times. These parameters are chosen based on Haswell specification parameters from the Intel manual [109, 111]. System memory is 2TB, with the workload inputs scaled so that each workload actually makes use of the full memory capacity.

Our cores maintain private L1 TLBs for different page sizes; i.e., 64-entry 4-way associative L1 TLBs for 4KB pages, 32-entry 4-way L1 TLBs for 2MB pages, and 4-entry TLBs for 1GB pages. As per Haswell specifications [111], our L1 TLBs are single-cycle and are accessed in parallel with the L1 caches using the standard virtually-indexed physically-tagged configuration [162]. All L1 TLBs have two read ports and a write port. Misses in the L1 TLB are followed by an L2 TLB lookup. Our baseline assumes the Intel Haswell configuration of private 1024-entry, 8-way associative L2 TLBs that can concurrently support 4KB and 2MB pages. In our studies, this baseline is 9 cycles based on post-synthesis SRAM numbers we generate, which also matches data from Intel manuals [111]. Our studies focus on varying this L2 TLB organization and latency; furthermore, we assume 2/1 read/write ports for each private L2 TLB and per shared L2 TLB slice. Finally, our simulator models the L2 TLBs accesses as being pipelined, so one request can be serviced every cycle. Finally, we combine our simulation framework with McPAT for our energy studies [140]. We model energy numbers for the cores, caches, etc.

Target Configurations: Table 3.2 details the shared L2 TLB configurations that we evaluate. The first approach we evaluate is the standard `monolithic` approach posed in the original shared L2 TLB study [37]. We have evaluated several banking configurations for `monolithic` and settle on 4 banks for 16- and 32-core configurations,

Table 3.2: Major configurations of TLB that were simulated.

	L2 TLB Entries (8-way associative)	**Physical Org**	**Interconnect**
Private	1024	1 TLB Per Core	-
Monolithic (Shared)	1024×NumCores	Monolithic	Mesh (Multi-Hop), SMART
Distributed (Shared)	1024×NumCores	1 slice Per Core	Mesh (Multi-Hop)
Nocstar	920×NumCores	1 slice Per Core	NOCSTAR

and 8 banks for 64 cores. We evaluate this with a regular mesh, and a single-cycle SMART NOC [132]. The second approach we study is a `distributed` approach where the shared L2 TLB is made up of an array of TLB slices placed near each core and connected by a NOC. We consider two different types of NOCs for shared distributed L2 TLBs: (a) *Mesh (Multi-hop):* This involves a traditional 1-cycle router coupled with 1-cycle link latency. To compete against a single-cycle-traversal-based NOCSTAR, we place enough buffers and links in the system to prevent link contention. Including any network contention may further degrade performance of workloads for traditional mesh networks. (b) NOCSTAR*:* A single cycle traversal if there is no contention; otherwise waits for another cycle as explained in Section 3.3.2; routing is XY-based. Our NOCSTAR evaluations assume that each core maintains a 920-entry (rather than a 1024-entry) shared TLB slice. This is a conservative area-normalized analysis, even though our interconnect consumes less than 1% area of each TLB slice.

Benchmarks: We use benchmarks from Parsec [40] and CloudSuite [82] for our studies. Furthermore, we study the performance of multi-programmed workloads by creating combinations of 4 applications. Each application in a multi-programmed workload has 8 threads executing and scaled up to use 2TB of memory.

3.5 Experimental Evaluations

Performance: Figure 3.11 shows performance results for a 16-core Haswell configuration, assuming only 4KB pages. We plot speedups versus a baseline with private L2 TLBs; i.e., higher numbers are better (note that the y-axis begins at 0.8). Our

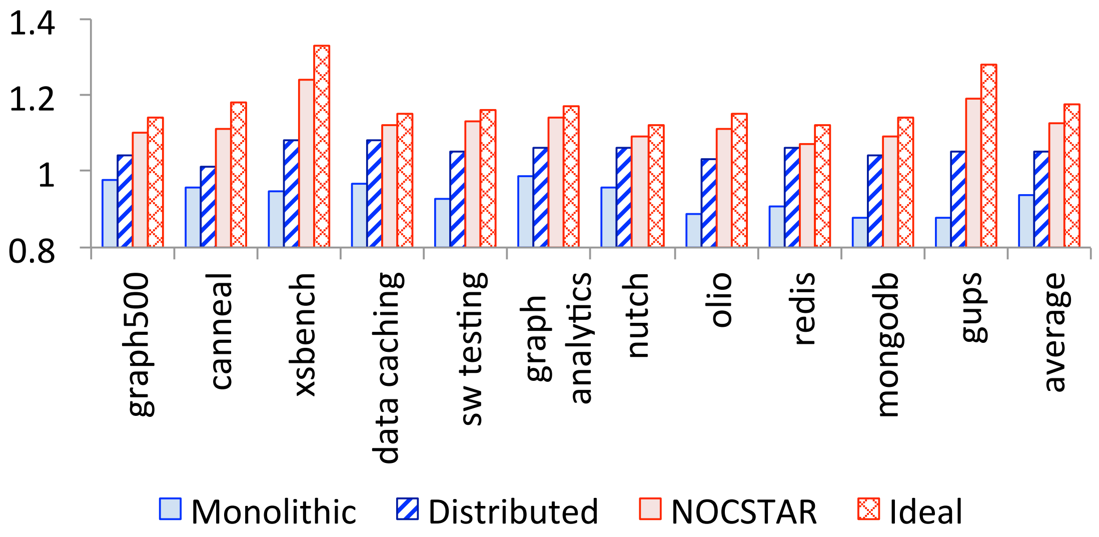


Figure 3.11: Speedups for monolithic, distributed, and NOCSTAR compared to ideal case with zero interconnect latency to the shared L2 TLB. Results assume 16-core Haswell systems using only 4KB pages.

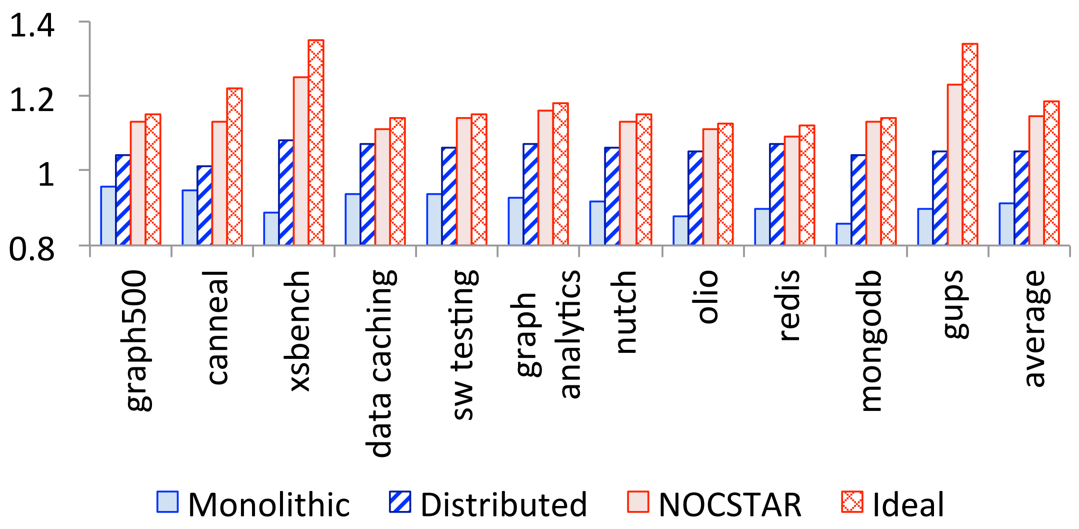
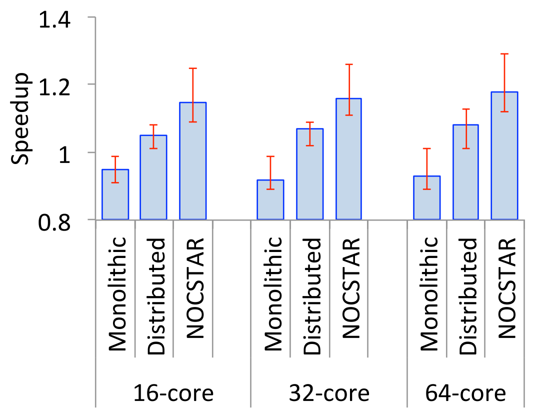


Figure 3.12: Complementary results to Figure 3.11 but when Linux uses transparent superpages for a mix of 4KB and 2MB pages.

`monolithic` data corresponds to a monolithic banked shared L2 TLB with access latencies determined from our circuit-level studies (see Section 3.4). We also show a `distributed` configuration as well an `ideal` case, where all shared TLB accesses have zero interconnect latency. Note that the `ideal` case does not imply an infinite TLB.

Figure 3.11 shows that NOCSTAR achieves an average of 1.13× and a max of 1.25× the performance of private L2 TLBs. Importantly, this is better than any other configuration. In fact, `monolithic` degrades performance versus private L2 TLBs because



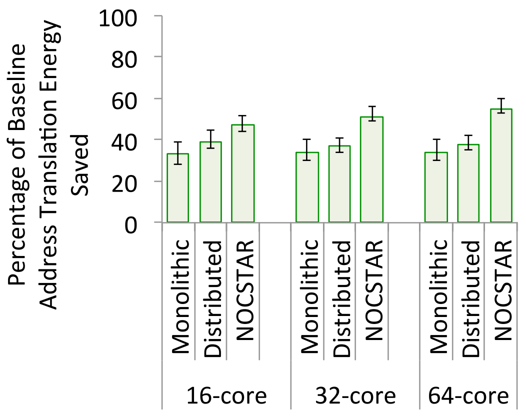


Figure 3.13: (Left) Speedups for varying core counts for Linux with transparent 2MB superpage support; and (right) percent of address translation energy saved versus private L2 TLBs.

of the perniciously high access latency. While `distributed` partly helps, Nocstar achieves over 8% additional performance and comes within 2% of `ideal`.

Figure 3.12 shows performance with Linux's native support for transparent 2MB superpages. We found that Linux was able to allocate 50-80% of each workload's memory footprint with superpages. One might expect superpages to reduce L1 TLB misses, reducing the gains from Nocstar. We find, however, *even better* performance with Nocstar in the presence of superpages. This is because the workloads are so memory-intensive (i.e., 2TB) that even with superpages, L1 TLB misses/shared L2 accesses are frequent. However, superpages do a good job of reducing shared L2 TLB misses, meaning that L2 TLB access times become a bigger contributor to overall performance. This explains why workloads such as `xsbench` and `gups` achieve large speedups of 1.2×+. Nocstar also outperforms `monolithic` and `distributed` with even larger margins than when simply using 4KB pages.

Scalability: The graph on the left in Figure 3.13 quantifies speedups for varying core counts, when Linux supports transparent 2MB superpages along with 4KB pages. We show average, minimum, and maximum speedup numbers. In the `monolithic` case, high hit rates are overshadowed by high access times, particularly worsening performance at higher core counts. Employing a `distributed` approach helps, but Nocstar consistently outperforms other approaches.

Energy: Recent work shows that address translation can constitute as much as 10-15% of overall processor power and that the energy spent accessing hardware caches for page table walks is orders of magnitude more expensive than the energy spent on TLB accesses [124]. Using a shared TLB saves address translation energy by eliminating a large fraction of page table walks. Figure 3.13 shows this, by plotting the percent of energy saved versus a baseline with private L2 TLBs. Even the `monolithic` approach eliminates roughly a third of address translation energy. However, NOCSTAR eliminates even more energy (as much as 60% on 64 cores). We have identified several reasons for these energy savings. One source is that NOCSTAR dramatically reduces runtime, thereby reducing static energy contributions of our system. Another important source of energy savings is that NOCSTAR reduces TLB misses and the ensuing page table walks. This means that cache lookups and memory references for the page table lookup are eliminated. In practice, like prior work [27, 28], we have found that most page table walk memory references are serviced from the LLC. In our experiments on a baseline without NOCSTAR 70-87% of the page table walks in the workloads we evaluate prompt LLC and main memory lookups for the desired page table entry. Using NOCSTAR eliminates the bulk – over 85% on average – of the LLC/memory references, thereby saving lookup energy. These energy savings far outweigh the the energy overheads of the dedicated NOCSTAR network.

Interconnect: We now tease apart the performance contributions of distributing TLB slices versus a faster interconnect with Figure 3.14. All bars represent speedups versus private L2 TLBs in a 32-core Haswell configuration. We show two versions of the banked monolithic approach, one with traditional multi-hop mesh, and one where we implement SMART with the monolithic approach. On average, both approaches suffer performance degradation; that is, even with a better interconnect (i.e., SMART), the monolithic approach experiences SRAM array latencies that are harmfully high. Instead, when we distribute the L2 TLB into slices per core (i.e., `distributed`), we achieve an average of 5% performance improvements. However, NOCSTAR performs even better.

Ideally, messages in NOCSTAR should take only 1 cycle to traverse the NOC. However, this number may increase because of contention for the path taken by the message.

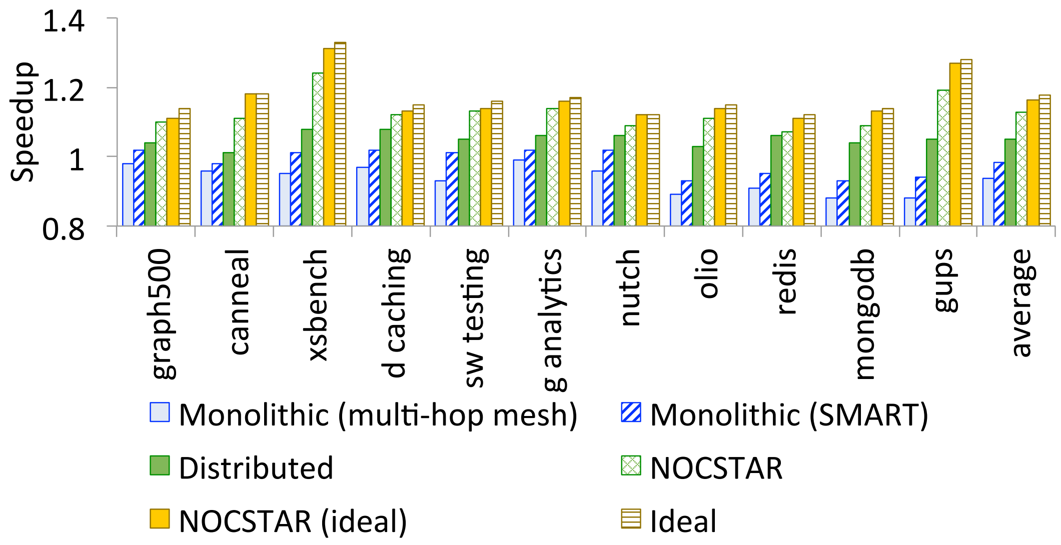


Figure 3.14: Speedup over baseline configuration with private L2 TLBs. We show two monolithic approaches (with traditional `multi-hop mesh` and `SMART`, as well as an `ideal` Nocstar, where we have no contention on the interconnect. We compare this to an `ideal` case where the TLB slices have zero interconnect latency.

We find that on average, latencies are 1-3 cycles, with only two workloads – `xsbench` and `gups` – suffering latencies that can go beyond 3 cycles. Overall, this means that Nocstar achieves performance close to an idealized case, where the interconnect faces zero contention (represented by `Nocstar (ideal)` in Figure 3.14. Finally, Figure 3.14 also shows the achievable performance with an `ideal` scenario where the interconnect has zero latency. We see that Nocstar achieves within 95% of the performance of this idealized case.

To test the interconnect mechanism adopted in Nocstar, we injected random synthetic traffic to a 64 core system. Figure 3.10(c) shows the average network latency faced by messages. Ideally messages in Nocstar would experience 1 cycle in path setup and another cycle to traverse the network. We see that even with an injection rate of 0.1 (1 message every 10 cycles per core, which is high for TLB traffic), the average latency of messages in the Nocstar interconnect remains within 3 cycles. Further, Figure 3.10(c) also shows the percentage of messages which experience no delay in acquiring a path.

Path setup options: We study two modes of link reservation: (a) Round trip acquire: links are acquired for the total period of accessing a remote slice. In this mode, link

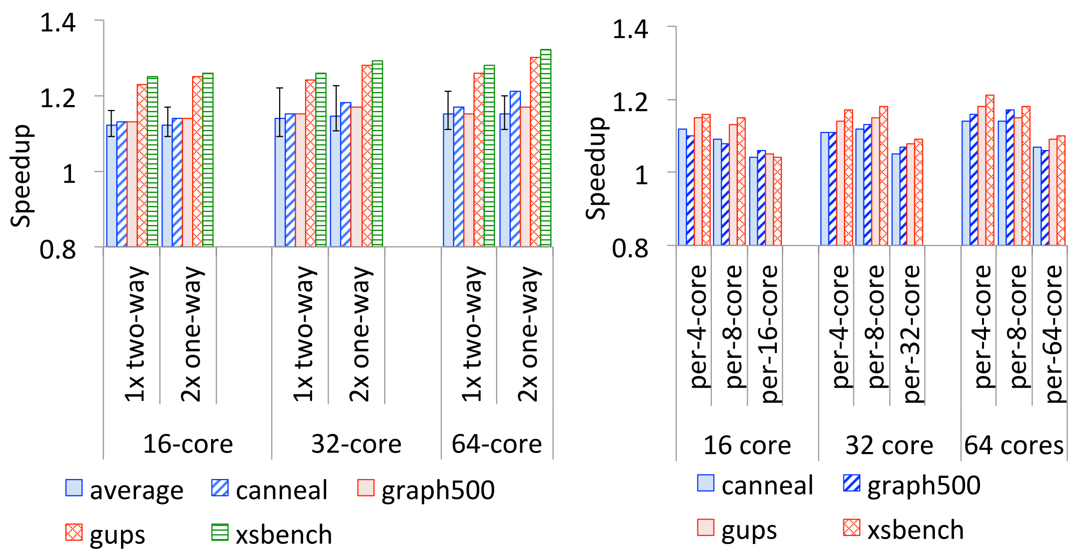


Figure 3.15: (Left) Speedups with varying core counts versus private L2 TLBs for round-trip acquire (`1×two-way`) and one-way acquire (`2×one-way`); and (right) speedups of TLB invalidation policies.

selection has to be performed only once for sending a request and response. (b) One-way acquire: Links are acquired only for sending a one-way message. Each message in the system selects links before traversal. The graph on the left in Figure 3.15 shows that acquiring links separately for each message delivers better performance than acquiring links for round trips.

TLB invalidation: We investigated the effect of sending an invalidate request to a TLB slice because of a shootdown or flush from any core. We considered various ways in which an invalidate message can be sent across a the TLB interconnect. The straightforward way is to send an invalidate from each core to the TLB slice. This policy is simple but may lead to congestion in the interconnect if all the cores are trying to invalidate from the same slice. The other way is to send the invalidate message to a central location which can then manage invalidations to all the slices. This can be further split up by having a manager for a set of n slices. The graph on the right in Figure 3.15 shows the speedup of workloads with different ways of sending an invalidate message compared to each core sending its own invalidate message.

Page table walk policies: We considered two policies for performing page table walks:

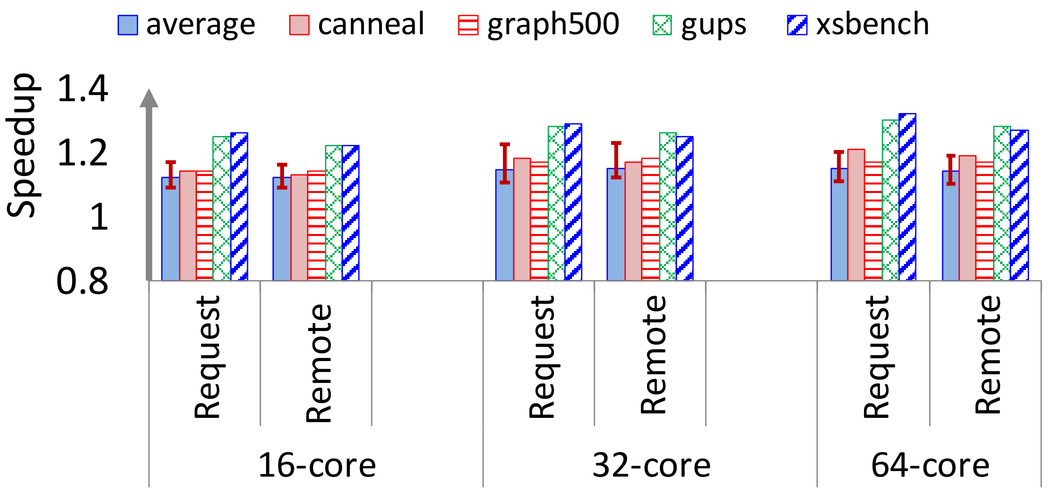


Figure 3.16: Page walks at requesting and remote core.

Page table walk at the remote core: The core which has the L2 slice for the virtual address performs the page walk and then sends the new translation as a response to the requesting core after inserting it in the L2 slice.

Page table walk at the request core: On an L2 TLB slice miss, a *miss* message is sent to the requesting core. The requesting core then performs the page table walk and sends an *insert* message to the remote slice.

Figure 3.16 shows speedups using policies. While performing the page table at the remote node involves sending fewer messages on the interconnect, it pollutes the local cache of the remote core (degrading performance). We see that performing the page table walk at requesting core delivers slightly better results compared to page table walk at remote core.

Sensitivity studies: We have quantified the NOCSTAR with other configurations (see Table 3.3). The first row quantifies the average and min/max speedups for our workloads for a 32-core Haswell. We compare this to scenarios with prefetching (Pref. column label), with hyperthreading (SMT column), and with varying page table walk latency (PTW Lat. column).

We first compare these numbers to a scenario where TLB prefetching is enabled. The original shared TLB paper studied the impact of prefetching translations ±1, 2, and 3 virtual pages adjacent to virtual pages prompting a TLB miss [37]. We run these

Pref.	SMT	PTW Lat.		Min	Avg	Max
None	1	Variable	Monolithic	0.89	0.92	0.99
			Distributed	1.02	1.07	1.09
			NOCSTAR	1.11	1.16	1.26
1	1	Variable	Monolithic	0.85	0.94	1.01
			Distributed	0.99	1.1	1.12
			NOCSTAR	1.08	1.2	1.29
1, 2	1	Variable	Monolithic	0.89	0.96	1.01
			Distributed	1.01	1.13	1.15
			NOCSTAR	1.1	1.25	1.32
1, 2, 3	1	Variable	Monolithic	0.87	0.89	0.99
			Distributed	0.99	1.08	1.11
			NOCSTAR	1.12	1.18	1.28
None	2	Variable	Monolithic	0.92	0.94	1.01
			Distributed	1.04	1.1	1.12
			NOCSTAR	1.14	1.21	1.31
None	4	Variable	Monolithic	0.93	0.95	1.03
			Distributed	1.01	1.13	1.15
			NOCSTAR	1.16	1.27	1.33
None	1	Fixed-10	Monolithic	0.84	0.88	0.93
			Distributed	0.94	0.95	0.99
			NOCSTAR	1.01	1.04	1.08
None	1	Fixed-20	Monolithic	0.89	0.92	0.99
			Distributed	1.02	1.07	1.09
			NOCSTAR	1.08	1.14	1.24
None	1	Fixed-40	Monolithic	0.93	0.97	1.03
			Distributed	1.05	1.09	1.13
			NOCSTAR	1.11	1.18	1.27
None	1	Fixed-80	Monolithic	1.05	1.08	1.12
			Distributed	1.08	1.13	1.17
			NOCSTAR	1.18	1.26	1.33

Table 3.3: Speedups for a 32-core Haswell system. We study the impact of prefetching, hyperthreading, and page table walk latencies on the speedups achieved by NOCSTAR and other shared L2 TLB configurations versus private L2 TLBs. Speedup averages across workloads, as well as minima/maxima are shown.
.

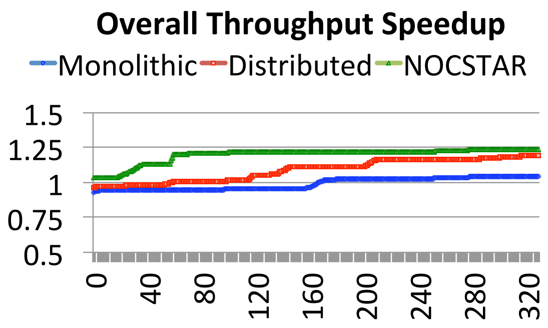
experiments with our monolithic, distributed, and NOCSTAR configurations in rows 2-4. We find that NOCSTAR's benefits are consistently enjoyed even in the presence of prefetching. Like the original shared TLB paper, we find that prefetching translations for up to ± 2 virtual pages away is most effective, with more aggressive prefetching polluting the TLB. However, in every one of these scenarios, the shared L2 TLB's bigger size implies that there is less pollution versus private L2 TLBs. Additionally, NOCSTAR's reduced access latency versus the monolithic and distributed approaches means that accurate prefetching can yield better performance.

Table 3.3 quantifies the impact of running multiple hyperthreads. The more the number of hyperthreads run per core, the higher the TLB pressure. As expected, this means that shared L2 TLBs offer hit rate benefits over private L2 TLBs; when combined with NOCSTAR's superior access latency, the performance exceeds distributed and monolithic results.

Finally, Table 3.3 quantifies NOCSTAR's performance as a function of the page table walk latency. We classify page table walk latency as variable (corresponding to a realistic simulation environment where the page table walk latency depends upon where in the cache the desired translations reside) or fixed-N (where we fix the page table walk latency to N cycles). As expected, when the page table walk latency is unrealistically low (i.e., 10 cycles), the monolithic and distributed TLBs *severely harm* performance. This is because these configurations suffer higher access latencies, while their higher hit rates are not useful because the impact of a TLB miss is minor. Nevertheless, even in this situation, NOCSTAR outpeforms private L2 TLBs. In more realistic scenarios where the page table walk latency is 20-40 cycles (which is what we typically find them to be on real systems [66, 37, 35, 33]), NOCSTAR's performance notably exceeds other options. And in scenarios where page table walks are very high (i.e., 80 cycles), these benefits become pronounced, with NOCSTAR outperforming distributed L2 TLBs by 13% on average.

Multiprogrammed combinations of sequential workloads: Our target platform is the 32-core Haswell system. Our workloads consist of combinations of four workloads, leading to 330 combinations overall. Each workload executes 8 threads to utilize all 32 cores. Figure 3.17 sorts our results by overall IPC improvement. NOCSTAR is particularly effective for multiprogramming because it offers the utilization benefits of shared TLBs without penalizing applications with high access latency. So, it *always* improves aggregate IPC compared to the other approaches; in contrast, `monolithic` degrades performance for about half the workloads because of access latency issues while `distributed` degrades 10% of the workloads.

The bottom graph in Figure 3.17 shows the speedup of the worst-performing application. As shown, `monolithic` and `distributed` see many cases (almost half the



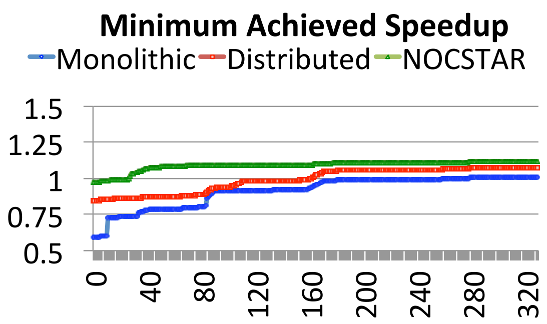


Figure 3.17: (Left) Overall throughput on 32 cores with 330 combinations of 4 workloads; and (Right) Speedup of the worst-performing sequential application over private L2 TLBs.

combinations) where at least one application suffers performance loss due to high access latency. Sometimes, degradation is severe; e.g., 40%. In contrast, only in 7% of the workloads does Nocstar degrade performance. Not only is this relatively rare, the extent of the performance loss is relatively benign, with worst cases of 2-3% versus private L2 TLBs. This problem is reminiscent of interference issues in LLCs and can likely be alleviated with LLC QoS/fairness mechanisms [187, 65]. We leave these for future work.

Pathological workloads: Our studies thus far suggest that most real-world workloads do not tend to generate significant congestion. For this reason, to stress-test Nocstar, we have devised two classes of microbenchmarks.

① TLB storm microbenchmark: The first microbenchmark triggers frequent context switches and page remappings. This forces "storms" of L2 TLB invalidations/accesses that congest the network. We take the workloads that we have profiled so far and we concurrently execute a custom-microbenchmark. We modify the Linux scheduler to context switch between our workloads and the microbenchmark; normally, Linux permits context switching at 10ms granularity, but we study unrealistically aggressive context switches from 0.5ms onwards for the purposes of stressing Nocstar. The custom microbenchmark is then designed to allocate 4KB pages, promote them to 2MB superpages, and then break then into 4KB pages again. The confluence of our modified Linux scheduler and our microbenchmark is a massive number of TLB misses and invalidations. Every context switch on our x86 Haswell systems forces all shared TLB contents to be flushed, followed by a storm of L2 TLB lookups for data. Furthermore,

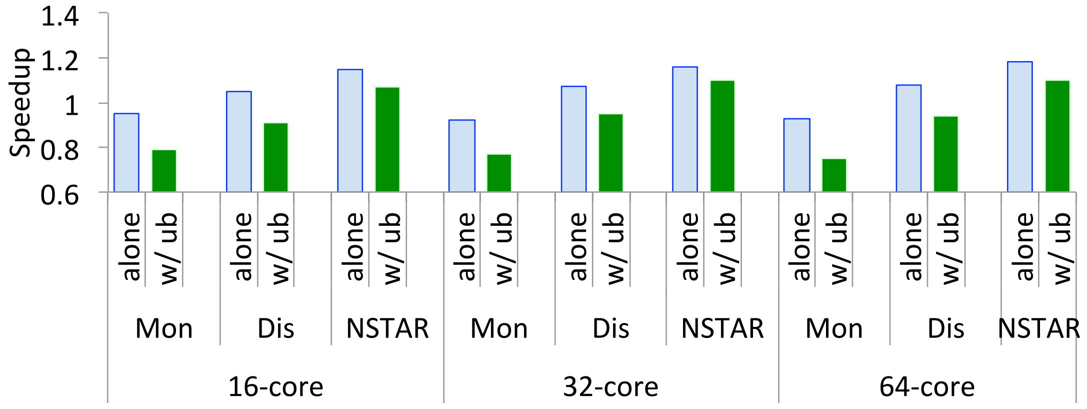


Figure 3.18: Average speedups for workloads versus private L2 TLB configuration, for varying core counts. Bars for `alone` represent results from when the workloads run alone (i.e., matching already-presented data). Bars for `w/ub` represent data for when the workloads were concurrently run with the TLB storm microbenchmark.

every time our microbenchmark promotes 4KB pages to a 2MB superpage, it invalidates 512 distinct L2 TLB entries.

Figure 3.18 quantifies the slowdown of our workload with this TLB activity. Results are averaged across all workloads and we vary core counts. We focus on the case which generates the maximum network congestion by context switching at 0.5ms; our microbenchmark generates as many as 200-300 L2 TLB accesses per kilo-instruction, which is more than the TLB stressmarks in prior work [169, 33].

Figure 3.18 shows that even the TLB pressure imposed by our microbenchmark naturally degrades performance versus the scenario where the benchmark is standalone (i.e., `alone`). As we can see, the `w/ub` results representing the microbenchmark suffer from as much as 10-20% performance degradation. However, *in ever single case*, Nocstar vastly outperforms the other approaches. For example, the `monolithic` banked L2 TLBs degrade performance by as much as 20-30% versus private L2 TLBs in the presence of this level of contention. On the other hand, Nocstar continues to achieve 7-11% performance improvements on average. While this is certainly lower than the 18%+ performance improvements achievable without congestion, these results are promising. Furthermore, the improvements achieved by Nocstar improve when we change our context switching granularity from an unreasonably aggressive 0.5ms to 1-10ms.

② TLB slice microbenchmark: We have also crafted a second microbenchmark to test what happens when there is immense congestion on one TLB slice. In this microbenchmark, we run *N-1* threads on our *N*-core machine. All these threads are designed to continuously access the L2 TLB slice assigned to the *Nth* core. Naturally, this approach degrades performance most severely. However, we find that in *every single case*, NOCSTAR continues to do better (by 3-5%) over private L2 TLBs. Furthermore, NOCSTAR is, in the most conservative scenario, 7% better than any other shared L2 TLB approach (i.e., either the monolithic banked, or distributed approaches). Consequently, NOCSTAR continues to be a better alternative than any other shared L2 TLB configuration.

3.6 Conclusions

The higher hit rate delivered by an SLL TLB is overshadowed by the high latency offered by the TLB structure and the network involved in traversing to it. Moreover, a traditional distributed architecture does not deliver the potential performance gains because of network latency. By co-designing distributed TLBs with a SMART interconnect, NOCSTAR multi-threaded and multi-programmed workload performance.

Chapter 4

Scheduling Page Table Walks for Irregular GPU Applications

4.1 Introduction

GPUs have emerged as a first-class computing platform. The massive data parallelism of GPUs had first been leveraged by highly-structured parallel tasks such as matrix multiplications. However, GPUs have more recently found use across a broader range of application such as graph analytics, deep learning, weather modeling, data analytics, computer-aided-design, oil and gas exploration, medical imaging, and computational finance [159]. Memory accesses from many of these emerging applications demonstrate a larger degree of *irregularity* – accesses are less structured and are often data dependent. Consequently, they show low spatial locality [55, 49, 151].

Irregular memory accesses can be particularly harmful to the GPU's Single-Instruction-Multi-Threaded (SIMT) execution paradigm where typically 32 to 64 threads (also called workitems) execute in a lockstep fashion (referred to as wavefronts or warps) [151, 210, 188, 180, 203, 211, 213]. When a wavefront issues a SIMD memory instruction (*e.g.,* load/store), the instruction cannot complete until data for all workitems in the wavefront are available. This is not a problem for well-structured parallel programs with regular memory access patterns where workitems in a wavefront typically access cache lines from only one or a few unique pages. The GPU hardware exploits this to gain efficiency by coalescing multiple accesses into a single access. For irregular applications, however, memory accesses of workitems within a wavefront executing the same SIMD memory instruction can access different cache lines from different pages. This leaves little scope for coalescing and leads to *memory access divergence* – *i.e.,* execution of a single SIMD instruction could require multiple cache accesses (when accesses fall on distinct cache lines) [180, 203, 211, 213] and multiple virtual-to-physical address translations (when accesses fall on distinct pages) [210, 172].

A recent study on real hardware demonstrated that such divergent memory accesses can slow down an irregular GPU application by up to 3.7-4× due to address translation overheads alone [210]. The study found that the negative impact of divergence could be greater on address translation than on the caches. Compared to one memory access on a cache miss, a miss in the TLB triggers a page table walk that could take up to four sequential memory accesses in the prevalent x86-64 or ARM architectures. Further, cache accesses cannot start until the corresponding address translation completes as modern GPUs tend to employ physical caches.

In this work, we explore ways to reduce address translation overheads of irregular GPU applications. While previous studies in this domain primarily focused on the design of TLBs, page table walkers, and page walk caches [175, 24, 172], we show that the *order in which page table walk requests are serviced* is also critical. We demonstrate that better scheduling of page table walks can speed up applications by 30% over a baseline first-come-first-serve (FCFS) approach. In contrast, naive random scheduling can slow applications down by 26%, underscoring the need of a *good* schedule for page table walks.

We observe that page walk scheduling is particularly important for a GPU's SIMT execution. An irregular application with divergent memory accesses can generate multiple uncoalesced address translation requests while executing a single SIMD memory instruction. For a typical 32-64 wide wavefront, execution of a single SIMD memory instruction by a wavefront can generate between 1 to 32 or 64 address translation requests. Due to the lack of sufficient spatial locality in such irregular applications, these requests often miss in TLBs, each generating a page table walk request. Furthermore, servicing a page table walk requires anything between one to four sequential memory accesses. Consequently, servicing address translation needs of a single SIMD memory instruction can require between 0 to 256 memory accesses. In the presence of such a wide variance in the amount of *work* (quantified by the number of memory accesses) required to complete address translation for an instruction, we propose a SIMT-aware page walk scheduler that prioritizes walk requests from instructions that would require less work. This aids forward progress by allowing wavefronts with less address translation traffic to complete faster.

Further, page walk requests generated by a single SIMD instruction often get interleaved with requests from other concurrently executing instructions. Interleaving occurs as multiple independent streams of requests percolate through a shared TLB hierarchy. However, in a GPU's SIMT execution model, it does not help a SIMD instruction to make progress if only a subset of its page walk requests is serviced. Therefore, servicing page walk requests in a simple first-come-first-serve (FCFS) order can impede the progress of wavefronts. Our proposed scheduler thus also *batches* requests from the same SIMD instruction for them to be serviced temporally together. The SIMT-aware scheduler speeds up a set of irregular GPU applications by 30%, on average, over FCFS.

To summarize, we make two key contributions:

- We demonstrate that *the order* of servicing page table walks significantly impacts the address translation overhead experienced by irregular GPU applications.
- We then propose a SIMT-aware page table walk scheduler that speeds up applications by up to 41%.

4.2 Background and the Baseline

This work builds upon two aspects of a GPU's execution: the GPU's Single-Instruction-Multiple-Thread (SIMT) execution hierarchy, and the GPU's virtual-to-physical address translation mechanism.

4.2.1 Execution Hierarchy in a GPU

GPUs are designed for massive data-parallel processing that concurrently operates on hundreds to thousands of data elements. To keep this massive parallelism tractable, a GPU's hardware resources are organized in a hierarchy. The top of Figure 4.1 depicts the architecture of a typical GPU.

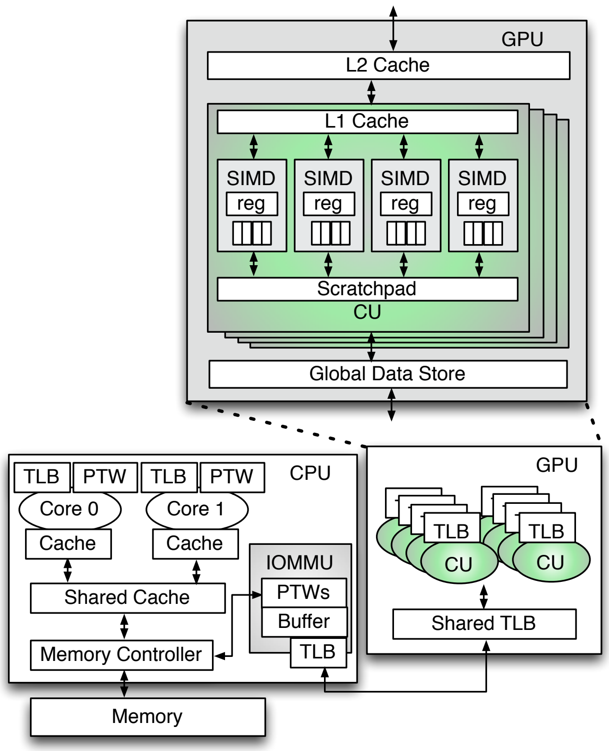


Figure 4.1: Baseline system architecture.

Compute Units (CUs) are the basic computational blocks of a GPU, and typically there are 8 to 64 CUs in a GPU. Each CU includes multiple Single-Instruction-Multiple-Data (SIMD) units, each of which has multiple *lanes* of execution (*e.g.,* 16). A SIMD unit executes a single instruction across all its lanes in parallel, but each lane operates on a different data item. A GPU's memory resources are also arranged in a hierarchy. Each CU has a private L1 data cache and a scratchpad that are shared across the SIMD units only within the CU. When several data elements accessed by a SIMD instruction reside in the same cache line, a hardware coalescer combines these requests into single cache access to gain efficiency. Finally, L1 caches are followed by an L2 cache that is shared across all CUs in a GPU.

GPGPU programming languages, such as OpenCL [92] and CUDA [1], expose to the programmer a hierarchy of execution groups that follows the hierarchy in the hardware resources. A workitem is akin to a CPU thread and is the smallest execution entity that runs on a single lane of a SIMD unit. A group of workitems, typically 32 to 64, forms a wavefront and is the smallest hardware-scheduled unit of work. All workitems in a wavefront execute the *same* SIMD instruction in a lockstep fashion but can operate on different data elements. An instruction completes execution *only* when *all* workitems in that wavefront finish processing their respective data elements. The next level in the hierarchy is the programmer-visible workgroup that typically comprises tens of wavefronts. Finally, work on a GPU is dispatched at the granularity of a kernel, comprised of several workgroups.

4.2.2 Virtual Address Translation in GPUs

As GPUs outgrow their traditional "co-processor" model to become first-class compute citizens, several key programmability features are making their way into mainstream GPUs. One such key feature is shared virtual memory (SVM) across the CPU and the GPU [13, 134]. For example, compliance with industry-promoted standards like the Heterogeneous System Architecture (HSA) requires SVM [134].

The bottom part of Figure 4.1 depicts the key hardware components of a typical SVM implementation in an HSA-enabled GPU. Conceptually, the key enabler for SVM in such a design is the GPU's ability to *walk* the same four-level x86-64 page table as the CPU. A page table is an OS-maintained data structure that maps virtual addresses to physical addresses at a page granularity (typically, 4KB). The IO Memory Management Unit (IOMMU) is the key component that enables a GPU to walk x86-64 page tables. While we focus this study on the SVM implementation mentioned above, our proposal is likely to be more broadly applicable to any GPU designs with virtual memory, not only to those supporting SVM. Next, we detail how an HSA-enabled GPU performs address translation.

GPU TLB Hierarchy: Just like in a CPU, the GPU's TLBs cache recently-used address translation entries to avoid accessing in-memory page tables on every memory access. Each CU has a private L1 TLB shared across the SIMD units. When multiple data elements accessed by a SIMD instruction reside on the same page, only a single virtual-to-physical address translation is needed. This is exploited by a hardware coalescer to lookup the TLB only once for such same page accesses. The GPU's L1

TLBs are typically backed by a larger L2 TLB that is shared across all the CUs in the GPU (bottom portion of Figure 4.1) [175, 172]. A translation request that misses in the GPU's TLB hierarchy is sent to the IOMMU [210]

IOMMU and Page Table Walkers: The IOMMU is the hardware component in the CPU complex that services address translation requests for accesses to the main memory (DRAM) by any device or accelerator, including that of the GPU [6, 7, 210]. The IOMMU itself has two levels of TLBs, but they are relatively small and designed to primarily serve devices that do not have their own TLBs (*e.g.,* NIC). The IOMMU's page table walkers, however, play an essential role in servicing the GPU's address translation requests. Upon a TLB miss, a page table walker *walks* an in-memory x86-64 page table to locate the desired virtual-to-physical address mapping.

An IOMMU typically supports multiple independent page table walkers (*e.g.,* 8-16) to concurrently service multiple page table walk requests (TLB misses) [210]. Multiple walkers are important for good performance because GPUs demand high memory bandwidth and consequently, often send many walk requests to the IOMMU.

The translation requests that miss in the TLB hierarchy queue up in the IOMMU's page walk request buffer (in short, IOMMU buffer). When a page table walker becomes free (*e.g.,* after it finishes servicing a page walk), it could start servicing a new request from the IOMMU buffer in the order it arrived. Later in this work, we will demonstrate that such an FCFS policy for selecting page table walk requests is not well-aligned with a GPU's SIMT execution model.

Another important optimization that the IOMMU borrows from the CPU's MMU design is the page table walk caches [210, 33, 27]. Nominally, a page table walk requires four memory accesses to walk an x86-64 page table, structured as a four-level radix tree. To reduce this, the IOMMU employs small caches for the first three levels of the page tables. These specialized caches are collectively called page walk caches (PWCs). Hits in PWCs can reduce the number of memory accesses needed for a walk to anything from one to three, depending upon which intermediate level produces the hit. For example, a hit for the entire top three levels will need one memory request to complete the walk by accessing only the leaf level. In contrast, a hit for only the root level requires three memory accesses. A complete miss in PWCs however, requires four accesses.

Putting it together: Life of a GPU Address Translation Request: ① An address translation request is generated when executing a SIMD memory instruction (load/store). ② A coalescer merges multiple requests to the same page (*e.g.,* 4KB) generated by the same SIMD instruction. ③ The coalesced translation request looks up the GPU's L1 TLB and then the GPU's shared L2 (if L1 misses). ④ On a miss in the GPU's L2 TLB, the request is sent to the IOMMU. ⑤ Upon arrival at the IOMMU, the request looks up the IOMMU's TLBs. ⑥ On a miss, the request queues up as a page walk request in the IOMMU buffer. ⑦ When an IOMMU's page table walker becomes free, it typically selects a pending request from the IOMMU buffer in FCFS order. ⑧ The page table walker first performs a PWC lookup and then completes the walk of the page table, generating one to four memory accesses. ⑨ On finishing a walk, the desired translation is returned to the TLBs and ultimately to the SIMD unit that requested it.

4.3 The Need for Smarter Scheduling of Page Table Walks

Irregular GPU applications often make data-dependent memory accesses with little spatial locality [55, 49]. This causes memory access divergence in the GPU's SIMT execution model where different workitems within a wavefront access data on distinct pages. The hardware coalescer is ineffective in such cases as several different address translation requests are generated by the execution of a single SIMD memory instruction. These requests then look up TLBs but often miss there owing to less locality in irregular applications. Eventually, many of these requests queue up in the IOMMU buffer to be serviced by a page table walker.

A recent study on commercial GPU hardware demonstrated that such divergent access can slowdown irregular GPU applications by up to 3.7-4× due to address translation overheads [210]. In this work, we aim to reduce address translation overheads for such irregular GPU applications.

We discover that the *order in which page table walks are serviced* can significantly

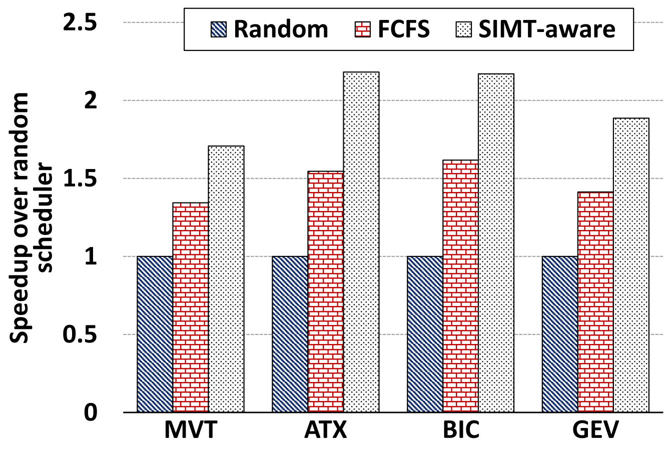


Figure 4.2: Performance impact of page walk scheduling.

impact the address translation overheads experienced by an irregular GPU application. While better page table walk scheduling (ordering) can potentially improve performance, poor scheduling (*e.g.,* random scheduling) can be similarly detrimental. Figure 4.2 shows the extent by which scheduling of page table walks can impact performance on a set of representative irregular applications (methodology is detailed in Section 4.5.1). The figure shows speedups of each application while employing naive random scheduler[1], the baseline FCFS, and the proposed SIMT-aware page walk scheduler. Each bar in the cluster shows the speedup of an application with a given scheduler, normalized to that with random scheduler. While we will detail our SIMT-aware scheduling over the next two sections, the key message conveyed by the figure is that the performance of an application can differ by more than 2.1× due to the difference in the schedule of page table walks. This underscores the importance of exploring the scheduling of a GPU's page walk requests.

A keen reader will notice the parallel between the scheduling of page table walks and the scheduling of memory (DRAM) accesses at the memory controller [177, 128, 23, 138]. The existence of a rich body of research on memory controller scheduling suggests that there exist opportunities for follow-on work to explore different flavors of page walk scheduling for both performance and QoS.

[1]As its name suggests, the random policy randomly picks a pending page walk request to service from the IOMMU buffer.

In the rest of this section, we discuss why page table walk scheduling affects performance and then provide empirical analysis to motivate better scheduling of GPU page walks.

4.3.1 Shortest-job-first Scheduling of Page Table Walks

We observe that instructions issued by a wavefront require different amounts of *work* to service their address translation needs. There are two primary reasons for this. First, the number of page table walks generated due to the execution of a single SIMD memory instruction can vary widely based on how many distinct pages the instruction accesses and the TLB hits/misses it generates. In the best case, all workitems in a wavefront access data on the same page and the perfectly coalesced translation request hits in the TLB. No page walks are necessary in that case. At the other extreme, a completely divergent SIMD instruction can generate page table walk requests equal to the number of workitems in the wavefront (here, 64). Second, each page walk may itself need anywhere between one to four memory requests to complete. This happens due to hits/misses in page walk caches (PWCs) that store recently-used upper-level entries of four-level page tables (detailed in Section 4.2).

Figure 4.3 shows the distribution of the number of memory accesses required to service address translation needs of SIMD instructions for a few representative applications. The x-axis shows buckets for the number of memory accesses needed by a SIMD memory instruction to service its address translation needs. The y-axis shows the fraction of instructions issued by the application that fall into the corresponding x-axis buckets. We excluded instructions that did not request any page table walks. We find that often between 27-61% of the instructions needed one to sixteen memory accesses to complete all the page table walks it generated. On the other hand, more than 33-70% of the instructions required forty-nine or more memory accesses. One of the applications (`GEV`) had close to 31% of instructions requiring sixty-five or more memory accesses. In summary, we observe that the amount of *work* (quantified by the number of memory accesses) required to service the address translation needs of an instruction varies significantly.

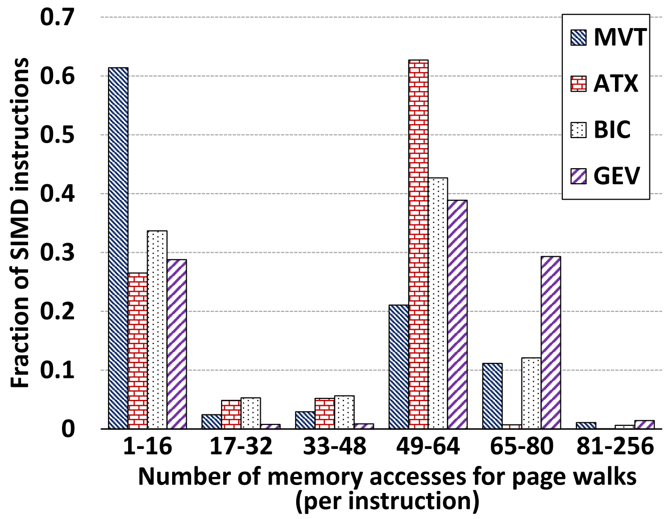


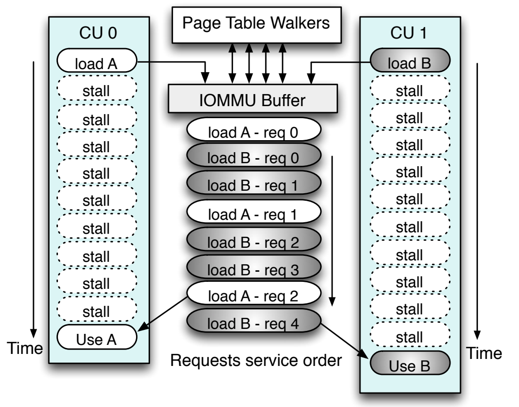
Figure 4.3: Distribution of number of memory accesses (*i.e.,* 'work') for servicing address translation needs of SIMD instructions.

It is well studied in scheduling policies across various fields that in the presence of "jobs" of different lengths, if a longer job can delay a shorter job, then it impedes overall progress. This leads to the widely employed "shortest-job-first" (SJF) policy that prioritizes shorter jobs over longer ones [10]. By analogy, we posit that servicing all page table walks generated due to the execution of a single instruction should be treated as a single "job" because the instruction cannot complete until all those walks are serviced. Figure 4.3 demonstrates that the "length" of such jobs, as quantified by the number of memory accesses, vary significantly.

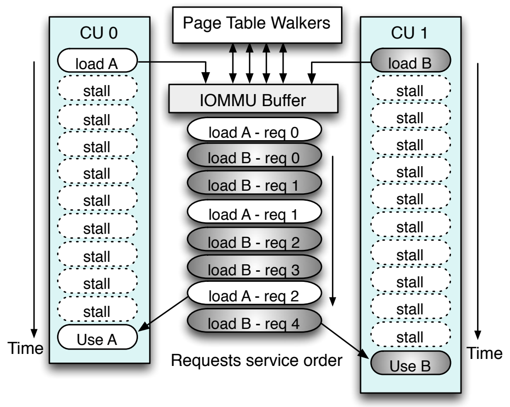
Key idea ①: Following the wisdom of time-tested SJF policies, we propose to prioritize the servicing of page table walk requests from instructions requiring fewer memory requests to complete their address translation needs over those requiring larger number of memory accesses.

4.3.2 Batch-scheduling of Page Table Walk Requests

Owing to the GPU's SIMT execution model, all page table walks generated by a single SIMD instruction must complete before the instruction can finish execution. The performance is thus determined by when the *last* of those walk requests is serviced. Even servicing all but one walk request does not aid progress.



(a) *Interleaving of page walk requests with hurts progress.*



(b) *Batching of page walk requests reduces interleaving.*

Figure 4.4: Impact of interleaving of a GPU's page walk requests.

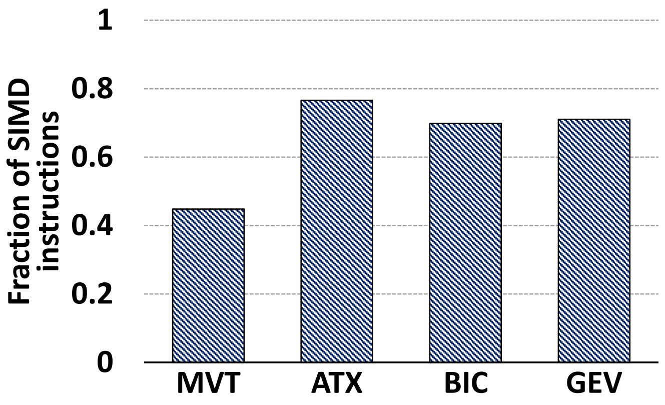


Figure 4.5: Fraction of instructions whose page walk requests are interleaved with requests from other instructions.

Figure 4.4(a) illustrates how the progress of two SIMD instructions, `load A` and `load B`, issued by two wavefronts are impaired if their page table walk requests are interleaved. Both `load A` and `load B` generate multiple walk requests and both experience stalls due to the latency to service walk requests generated by the other. Evidently, if interleaved page walk requests are serviced in the FCFS order, then it delays completion of both `load A` and `load B` since both need all their walk requests to finish before the instruction can progress. This inefficiency exacerbates if walk requests from a larger number of distinct instructions interleave since the progress of every instruction involved in the interleaving suffer.

Unfortunately, such interleaving among page walk requests from different SIMD

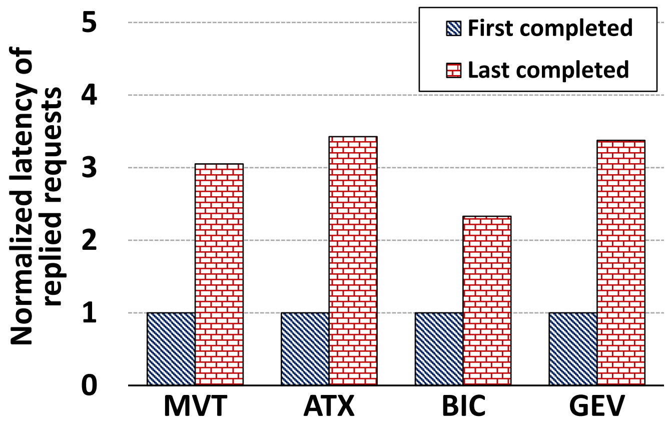
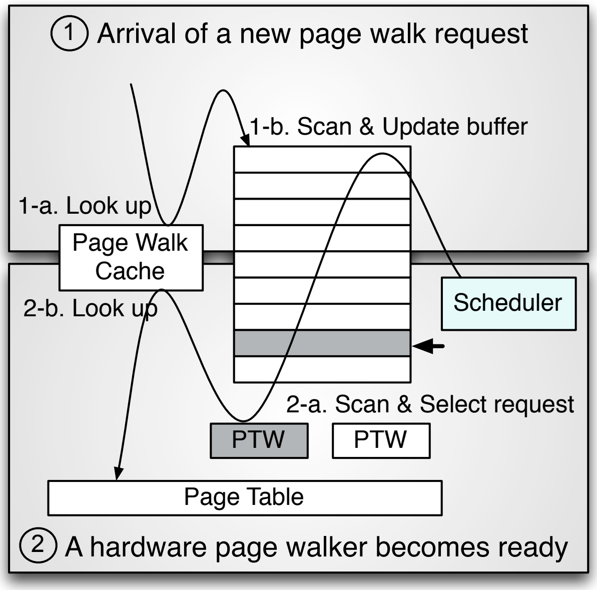


Figure 4.6: Average latencies of the first- and the last-completed page walk request per instruction.

instructions happens fairly regularly. Figure 4.5 quantifies how often such interleaving happens for representative irregular GPU workloads (methodology detailed in Section 4.5.1). The y-axis shows the fraction of executed memory instructions whose page walk requests interleave with requests from at least another instruction. We exclude any instructions that do not generate at least two page table walks as interleaving is impossible for them. We observe that 45-77% of such instructions have their walk requests interleaved.

We traced the source of this interleaving to the GPU's shared L2 TLB. The shared L2 TLB receives multiple independent streams of address translation requests generated by L1 TLB misses from concurrently executing wavefronts across different CUs. These requests can then miss in the L2 TLB and travel to the IOMMU. The IOMMU thus receives a multiplexed stream of walk requests from different wavefronts.

A reasonable question to ask is how much does this interleaving potentially impact the performance? Figure 4.6 shows the potential performance cost of interleaving. The figure shows the average latencies experienced by the first- and the last-completed page walk requests from the same SIMD memory instruction. The latencies are normalized to the average latency experienced by the first completed walk request. We exclude instructions that do not generate at least two page walk requests as they cannot interleave. Larger the latency gap, the more time an instruction potentially stalls for all of its page walk requests to complete. We observe that often the latency of the last



① Arrival of a new page walk request
1-b. Scan & Update buffer
1-a. Look up
Page Walk Cache
2-b. Look up
Scheduler
2-a. Scan & Select request
PTW
PTW
Page Table
② A hardware page walker becomes ready

Figure 4.7: Key components and actions of the SIMT-aware page table walk scheduler.

completed walk is more than 2-3× that of the first completed page walk. This suggests that the interleaving of page walks can significantly impede forward progress.

Ideally, page walk requests should be scheduled to minimize such latency gaps. A smarter scheduler thus should strive to achieve a schedule as shown in Figure 4.4(b) by batching page walk requests from the same instruction. We see from the figure that `load A` can potentially complete much earlier without further delaying `load B` in Figure 4.4(a).

Key idea ②: A smart scheduler should *batch* page walk requests from the same instruction to minimize interleaving due to walk requests from other instructions.

4.4 Design and Implementation

Driven by the above analyses, we propose a *SIMT-aware page table walk scheduler* in the IOMMU. Figure 4.7 shows some of the key components and actions in the IOMMU to realize such a scheduler. When an IOMMU's page table walker becomes available to accept a new request, the scheduler selects which pending page walk request is serviced next. While we introduce a specific scheduler design, there could be several other potential designs that build upon our observations about the importance of page table walk scheduling.

Our proposed SIMT-aware scheduler follows the two key ideas mentioned in the previous section. At a high level, the scheduler first attempts to schedule a pending page walk request (in the IOMMU buffer) issued by the same SIMD instruction as the most recently scheduled page walk. If none exists, it schedules a request issued by an instruction that is expected to require the least number of memory requests (*i.e., work*) to service all its walk requests. For this purpose, we assign a score to each page walk request. This score estimates the number of memory requests that would be required to complete *all* page walk requests of the issuing instructions. The score is thus the same for all pending page walk requests generated by a given SIMD instruction. A lower value indicates fewer memory requests to service an instruction's page walk requests.

We make a few simple hardware modifications to realize the above design concept. First, each page walk request from the GPU is attached with an instruction ID (20 bits in our implementation). Correspondingly, the buffer holding the pending page walk requests at the IOMMU is extended with this ID. As shown in Figure 4.7, we then modify how the IOMMU behaves when ① a new page walk request arrives at the IOMMU, and when ② a hardware page walker becomes available to accept a new request. Below we detail actions taken during these two events.

① Arrival of a new page walk request: If there is an idle hardware page walker when a new request arrives then it starts walking immediately. Otherwise, we assign an integer score (between 1 to 256, where 256 corresponds to the maximum possible number of memory accesses required if all 64 workitems need four memory accesses each to perform their respective translation) to the newly-arrived request. The score estimates the number of memory accesses needed to complete all page walk requests of the corresponding instruction. This is done in two steps. First, the new request looks up the PWCs to estimate the number of memory requests that this request alone may need to get serviced (action `1-a` in Figure 4.7). This number can be between one (on a hit in all upper-levels in the PWC and thus, requiring only single memory access to the leaf-level of the page table) to four (on a complete miss in the PWC requiring the full walk of the four-level page table). Since the PWC contents could change by the time the scheduler selects the request, this number is an estimate of the actual number of memory accesses required to service the walk.

Second, we then scan all the pending page walk requests in the IOMMU buffer to find any matching page walk requests issued by the same instruction as the newly arrived one (`1-b`). All requests from the same SIMD instruction have the same score. A new score is computed by adding the PWC-based score of the newly-arrived request to the previous score of an existing request in the IOMMU buffer that is issued by the same instruction. This updated score now represents the total estimated number of memory accesses required to service *all* the translation requests from the issuing SIMD instruction. All entries in the IOMMU that match the SIMD instruction of the newly-arrived request (including the newly-arrived request) are then updated with this new score.

② A hardware page walker becomes ready: When a page walk finishes, the corresponding page table walker becomes available to start servicing a new request. The scheduler decides which of the pending page walk requests (if any) it should service next. First, the scheduler scans the buffer of pending page walks to find any request that matches the instruction ID of the most recently issued page walk request (`2-a`). If such a request exists, it is chosen to ensure temporal batching of page walks issued by the same SIMD instruction. If no such matching request is found, then the scheduler selects a request with the lowest score. This follows the second key idea – schedule requests from the instruction that is expected to require the fewest memory accesses. Both actions are performed during the scanning of pending page walk requests (`1-a`). Finally, the selected page walk request is serviced as usual by first looking up the PWC for partial hits and then completing the walk of the page table (`2-b`).

Putting it all together: To summarize, an address translation from the GPU flows as follows. The coalescing and lookups in the GPU TLBs happen as before (refer to Section 4.2.2 for steps). The only modification for our scheduler is that each request now carries the ID of the instruction that generated it. As in the baseline, a translation request that misses in the GPU TLBs is sent to the IOMMU where it performs a lookup in the IOMMU's TLBs. If the request misses in all the TLBs, then it is inserted into the IOMMU buffer. If any of the page table walkers (8 in the baseline) are available, then one of them starts the page walk process. Otherwise, our scheduler calculates a score for the newly-arrived request and re-scores any of the already-pending requests from the same instruction as detailed above (actions `1-a` and `1-b`). The request then waits in the buffer until it is selected by the scheduler.

When a page table walker finishes a walk, the scheduler selects which request it should service next. The scheduler scans pending requests in the IOMMU buffer (action `2-a`) to find if there are any requests issued by the same instruction as the last-scheduled request. If so, the oldest among such requests is chosen. If not, the scheduler selects the request with the lowest score (oldest first in the case of a tie). Once a request is scheduled, the page table walker proceeds walking as usual – it looks up the PWC for partial hits before making memory accesses to the page table (`2-b`).

Design Subtleties: We now discuss a few intricacies of this design. First, note that the scanning of the pending page walk requests upon arrival of a new request is not in the critical path. The newly arrived request anyway queues up in the IOMMU buffer for the scheduler to select it. If a free page table walker is immediately available, the scheduler plays no role and no scanning is involved. However, it adds an extra latency in the critical path of servicing a new page walk request when the scheduler scans the pending request (`2-a`). Every such request in the IOMMU buffer has already suffered a long latency miss through the entire TLB hierarchy, and a walk itself requires hundreds of cycles. Therefore, the latency of scanning pending requests adds little additional delay.

As with any scheduler, the above design is susceptible to starvation. We implement an aging scheme whereby we prioritize pending walk requests that have been passed by a large number of younger requests (in our experiments, we found that setting this threshold to two million requests worked well to avoid any potential starvation).

As briefly mentioned earlier, another subtlety is that the PWC contents may change between the time a request arrives at the IOMMU and the time the scheduler selects that request. This could lead to inaccuracies in estimating the number of memory accesses needed to service a page walk since the score is calculated when the request arrives. Unfortunately, it is infeasible for the scheduler to re-calculate scores of every pending request at the time of request selection. This would have added significant latency in the critical path. Instead, we reduce this potential inaccuracy by adding 2-bit saturating counters to the entries of the PWC. Whenever a lookup for a newly-arrived request hits in the page walk cache (`1-a` in Figure 4.7), the counters of the corresponding entries are incremented. The counters are decremented when a selected page walk request hits in the PWC (`2-b`). Thus, a value greater than zero indicates that there exists at least one pending page walk request in the IOMMU buffer that would later hit in the page walk cache when that request is scheduled. The replacement policy in the page walk cache is then modified to avoid replacing an entry with a counter value greater than zero. If all entries in a set have value greater than zero, then a conventional pseudo-LRU policy selects a victim as usual.

4.5 Evaluation

We now describe our evaluation methodology and then analyze the results in detail.

4.5.1 Methodology

We used the execution-driven gem5 simulator that models a heterogeneous system with a CPU and an integrated GPU [4]. We heavily extended the gem5 simulator to incorporate a detailed address translation model for a GPU including coalescers, the GPU's TLB hierarchy, and the IOMMU. Inside the newly-added IOMMU module, we model a two-level TLB hierarchy, multiple independent page table walkers, and page walk caches to closely mirror the real hardware. We implemented different scheduling policies for page table walks, including our novel SIMT-aware page walk scheduler inside the IOMMU module.

The simulator runs unmodified applications written in OpenCL [92] or in HCC [53]. Table 4.1 lists the relevant parameters for the GPU, the memory system, and the address translation mechanism of the baseline system. Section 4.5.2 also presents sensitivity

Table 4.1: The baseline system configuration.

GPU	2GHz, 8 CUs, 4 SIMD per CU 16 SIMD width, 64 threads per wavefront
L1 Data Cache	32KB, 16-way, 64B block
L2 Data Cache	4MB, 16-way, 64B block
L1 TLB	32 entries, Fully-associative
L2 TLB	512 entries, 16-way set associative
IOMMU	256 buffer entries, 8 page table walkers 32/256 entries for IOMMU L1/L2 TLB, FCFS scheduling of page walks
DRAM	DDR3-1600 (800MHz), 2 channel 16 banks per rank, 2 ranks per channel

Table 4.2: GPU benchmarks for our study.

	Benchmark (Abbrev.)	Description	Memory Footprint
Irregular applications	XSbench (XSB)	Monte Carlo neutronics application	212.25MB
	MVT (MVT)	Matrix vector product and transpose	128.14MB
	ATAX (ATX)	Matrix transpose and vector multiplication	64.06MB
	NW (NW)	Optimization algorithm for DNA sequence alignments	531.82MB
	BICG (BCG)	Sub kernel of BiCGStab linear solver	128.11MB
	GESUMMV (GEV)	Scalar, vector and matrix multiplication	128.06MB
Regular application	SSSP (SSP)	Shortest path search algorithm	104.32MB
	MIS (MIS)	Maximal subset search algorithm	72.38MB
	Color (CLR)	Graph coloring algorithm	26.68MB
	Back Prop. (BCK)	Machine learning algorithm	108.03MB
	K-Means (KMN)	Clustering algorithm	4.33MB
	Hotspot (HOT)	Processor thermal simulation algorithm	12.02MB

studies varying key parameters.

Table 4.2 lists the applications used in our study with descriptions of each workload and their respective memory footprints. We draw applications from various benchmark suites including Polybench [174] (`MVT, ATAX, BICG,` and `GESUMMV`), Rodinia [58]

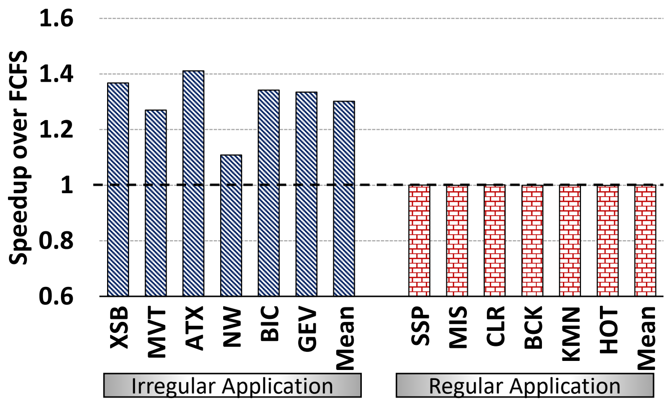


Figure 4.8: Speedup with SIMT-aware page walk scheduler.

(`NW, Back propagation, K-Means,` and `Hotspot`), and Pannotia [57] (`SSSP, MIS,` and `Color`). In addition, we used a proxy-application released by the US Department of Energy (`XSBench` [205]).

In this work, we focus on emerging GPU applications with irregular memory access patterns. These applications demonstrate memory access divergence [55, 49] that can bottleneck a GPU's address translation mechanism [210]. However, not every application we studied demonstrates irregularity nor suffers from significant address translation overheads. We find that six workloads (`XSB, MVT, ATX, NW, BCG,` and `GEV`) demonstrate irregular memory access behavior while the remaining workloads (`SSP, MIS, CLR, BCK, KMN,` and `HOT`) have fairly regular memory accesses. Applications with regular memory accesses show little translation overhead to start with and thus, offer little scope for improvement. Our evaluation thus focuses on applications in the first category, but we include results for the regular applications to demonstrate that our proposed techniques do not harm workloads that are insensitive to translation overheads.

4.5.2 Results and Analysis

We evaluate the impact of page table walk scheduling and our SIMT-aware scheduler by asking the following questions: ① How much does the SIMT-aware page table walk scheduler speed up applications over the baseline FCFS scheduler? ② What are the sources of speedups (if any)? ③ How sensitive are the results to configuration parameters like the TLB size and the number of page table walkers?

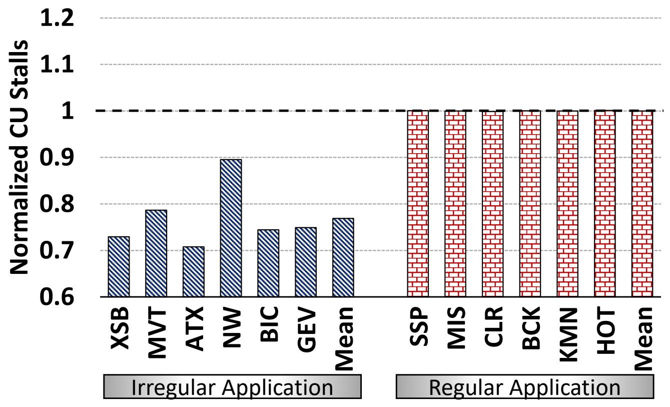


Figure 4.9: GPU stall cycles in execution stage.

Figure 4.8 shows the speedups of GPU applications with our SIMT-aware page walk scheduler over FCFS. The left half of the figure (dark bars) shows the speedups for irregular applications while the right half (thatched bars) shows the speedups for applications with regular memory accesses. We observe that our scheduler speeds up irregular GPU applications by up to 41%, and by 30% on average (geometric mean). On the other hand, there is little change in the performance of regular applications. This is expected; regular applications experience little address translation overhead, and thus page table walk scheduling has almost no influence on their performance. The data, however, assure that the SIMT-aware scheduling does not hurt regular workloads.

Previously in Figure 4.2 in Section 4.3, we also demonstrated how naive random scheduling can significantly hurt performance. Together, these observations show that ① different scheduling of page walks can have severe performance implications, and ② the SIMT-aware scheduler can significantly speed up irregular GPU applications without hurting others.

Analyzing Sources of Speedup

It is important to understand the reasons behind the observed speedups. Toward this, we first present how schedulers impact GPU stall cycles, which are the cycles during which a CU cannot execute any instructions because none are ready. Figure 4.9 shows the normalized stall cycles for each application with our SIMT-aware page table walk scheduler. The height of each bar is normalized to the stall cycles with the FCFS

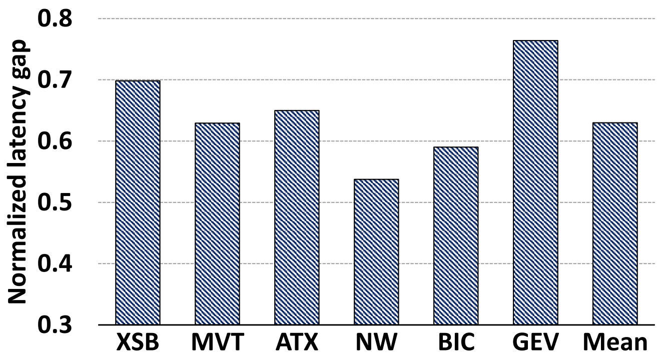


Figure 4.10: Latency gap between the first and the last-completed page walk request per instruction.

scheduler. A lower number indicates better forward progress since CUs are stalled for less time on average. As before, the left half shows the results for irregular applications and the right half shows those for regular applications. We observe that the SIMT-aware scheduler reduces the stall cycles by 23% on average (up to 29%) for irregular applications. This shows how the scheduler enables instructions, and consequently, corresponding wavefronts, to make better forward progress. This ultimately leads to faster execution. As expected, the stall cycles for regular applications remain mostly unchanged. Because these applications neither alter performance nor provide any new insights, the remaining evaluations in this Chapter focus entirely on irregular applications.

In Figure 4.6 (Section 4.3), we showed that there can be a significant gap between the latency of the first- and the last-completed page walk for a given SIMD instruction. A larger gap indicates that instructions are waiting for a large number of translation requests to be completed, or a few requests to be completed but that are delayed due to the servicing requests from other instructions, or a combination of both effects. Our SIMT-aware scheduler batches the servicing of page table walk requests from the same instruction to reduce this gap. Figure 4.10 shows the degree of effectiveness of our scheduler in reducing the gap. Each bar represents the latency gap between the first- and the last-completed page walk requests from an instruction with our scheduler. The height of each bar is normalized to the latency gap with the baseline FCFS scheduler. As before, we exclude instructions that generate less than two page table walks as they

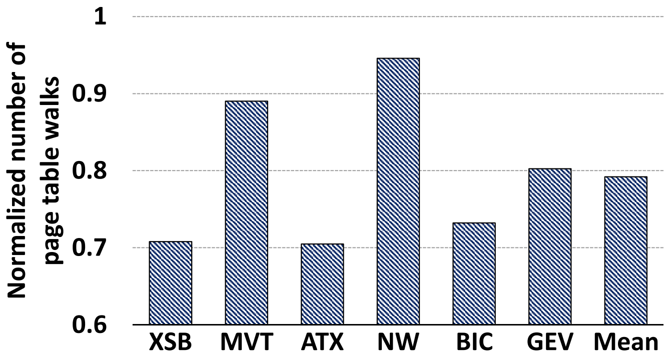


Figure 4.11: Number of page walk requests with SIMT-ware scheduler normalized over FCFS.

cannot interleave. We observe that the SIMT-aware scheduler reduces the latency gap by 37% over FCFS, on average. This shows the efficacy of batching page walks.

Another interesting performance impact of our scheduler is that it also reduced the total number of page table walk requests. Figure 4.11 shows the number of page walk requests (*i.e.,* number of TLB misses) with our SIMT-aware scheduler, normalized to the baseline FCFS scheduler.

We observed 21% reduction (up to 30%) in the number of page table walk requests, on average. We traced the reason for this improvement to the better exploitation of intra-wavefront locality in TLBs. Our scheduler favors SIMD instructions with lower address translation needs, which in turn aids forward progress. At the same time, our scheduler also tends to delay page walk requests from instructions that generate a large amount of address translation traffic. These high-overhead instructions are anyway likely to take a long time to complete. While the translation-heavy instructions are stalled, they are kept away from polluting (thrashing) the GPU's TLBs. Consequently, the low-overhead instructions experience higher TLB hit rates as the useful TLB entries are not evicted by the high-overhead instructions. This results in a reduction in the number of TLB misses and thus, reduce the number of page walk requests.

We further validated the above conjecture by counting the number of distinct wavefronts that access the GPU's L2 TLB over fixed-sized epochs (we used an epoch length of 1024 GPU L2 TLB accesses). Figure 4.12 presents this metric (normalized to FCFS), averaged over all epochs for the SIMT-aware scheduler. We observed a 42% reduction

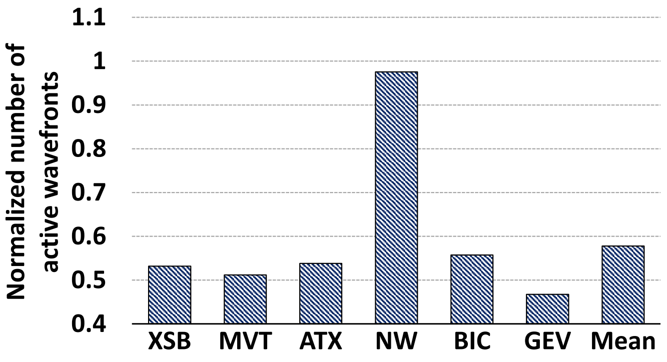
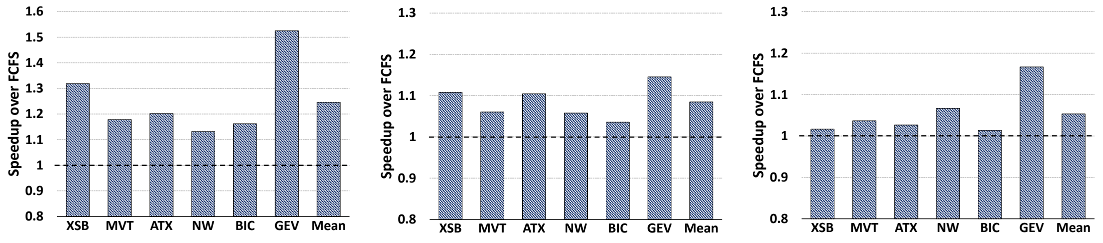


Figure 4.12: Number of active wavefronts accessing the GPU's L2 TLB with SIMT-aware scheduler (normalized over FCFS).



(a) *1024 L2 TLB and 8 walkers.* **(b)** *512 L2 TLB and 16 walkers.* **(c)** *1024 L2 TLB and 16 walkers.*

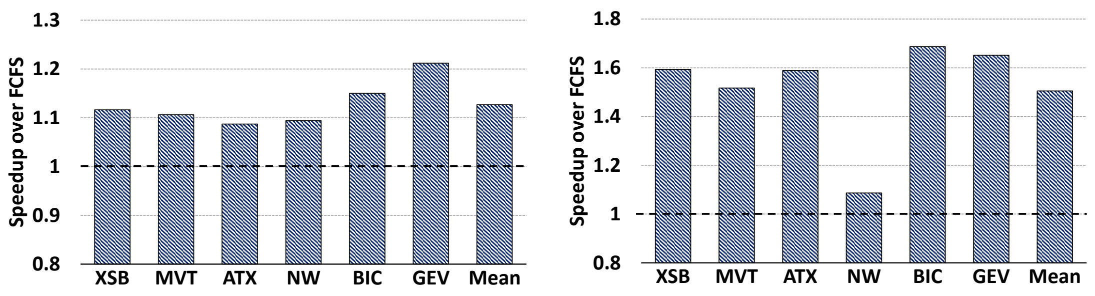
Figure 4.13: Speedups with varying GPU L2 TLB size and page table walker counts.

in the number of distinct wavefronts accessing the GPU's L2 TLB in an epoch. This shows the role of page walk scheduling in lowering the contention in the GPU's L2 TLB. Consequently, the number of page table walks decreases due to less potential thrashing in the TLB. This behavior has similarities to phenomena observed by others in the context of the GPU's caches [139].

Sensitivity Analysis

We measured sensitivity of the scheduler to the GPU's L2 TLB size, the number of concurrent page table walkers, and the size of IOMMU buffer holding the pending page walk requests.

Figure 4.13 shows the speedup achieved by our SIMT-aware scheduler with varying amounts of critical address translation resources: L2 TLB capacity and the number of page table walkers. Figure 4.13(a) shows the speedup with 1024 entries in L2 TLB and eight page table walkers. The average speedup achieved by the SIMT-aware scheduler over the FCFS scheduler is significant (on average, 25%) even with larger TLB. It is,



(a) *128 IOMMU buffer entries.* **(b)** *512 IOMMU buffer entries.*

Figure 4.14: Speedups with varying IOMMU buffer size.

however, slightly less than 30% speedup achieved with 512-entry L2 TLB. The larger TLB reduces the number of page walk requests and thus, the scope for improving performance by scheduling page walks diminishes.

On the other hand, Figure 4.13(b) shows the speedups with 16 page table walkers. Increasing the number of page table walkers reduces the number of pending page table walks as the effective address translation bandwidth increases. This also reduces head-room for the performance improvement achievable through better page walk scheduling. We observe that SIMT-aware page walk still speeds up applications by about 8.4% over the FCFS policy. Finally, Figure 4.13(c) shows the combined impact of both the bigger TLB size and the increased page table walker count. In this configuration, both the increased TLB resources and the increased number of page table walkers further moderate scope for the improvement with smarter scheduling of walks. SIMT-aware scheduling speeds up applications by 5.3% in this configuration.

Overall, our SIMT-aware scheduler consistently performs better than the baseline FCFS scheduler across different configurations and different workloads, thereby demonstrating the robustness of our technique, although the amount of benefit depends on the severity of address translation bottleneck.

We then investigate the effect of IOMMU buffer size on our scheduler. The IOMMU buffer size determines the size of the *lookahead* for the scheduler, *i.e.,* the maximum number of page walk requests from which it can select a request. Larger the buffer size, the larger is the lookahead potential. Figures 4.14(a) and 4.14(b) show speedups with SIMT-aware scheduler over the FCFS scheduler with 128-entry and 512-entry

IOMMU buffers, respectively. All other parameters remain the same as in the baseline configuration. A smaller IOMMU buffer size reduces the opportunity for a scheduler to make smart reordering decisions, and thus, the speedups due to SIMT-aware scheduling are reduced to 13% (Figure 4.14(a)) with a 128-entry buffer. On the other hand, if the size of the buffer is increased to 512 entries, the average speedup jumps to 50% (Figure 4.14(b)). In short, the magnitude of the performance benefit from SIMT-aware scheduling varies across configurations but remains substantial across all cases.

4.6 Discussion

Why not large pages? Large pages map larger ranges of contiguous virtual addresses (*e.g.,* 2MB) to contiguous physical addresses. They can reduce the number of TLB misses by mapping more of memory with the same number of TLB entries. However, large pages are far from a panacea as decades of deployment and studies in the CPU world have demonstrated [29, 84, 123]. As memory footprints of applications continue to grow, today's large page effectively becomes tomorrow's small page. Thus, techniques that help improve performance with small (base) pages remain useful for future workloads with larger memory footprints, even with larger page sizes. Even workloads with memory footprints of a few hundred MBs (Table 4.2) can benefit significantly from our SIMT-aware page walk scheduler, and workloads with more realistic footprints will continue to benefit from more efficient page walk scheduling, even with large pages. Unfortunately, exorbitant simulation time prevents us from evaluating such large memory footprint.

More importantly, irregular GPU applications tend to exhibit low spatial locality [55, 49] where large pages tend to have limited benefits. These applications see less benefit from large pages because the approach fundamentally relies on locality to enhance the reach of TLBs [29]. Previous works further demonstrated that large pages can even hurt performance in some cases due to the relatively lower number of entries in large page TLBs [66, 29]. Recent works on GPUs have also demonstrated that large pages can significantly increase the overhead of demand paging for GPUs [216, 24].

Interactions with Other Schedulers: In a GPU, wavefront (warp) schedulers play an important role in leveraging parallelism and impact cache behavior [179, 180]. Previous work has also shown the importance of TLB-aware wavefront scheduling [172]. Apart from the wavefront scheduler, memory controllers sport sophisticated scheduling algorithms to improve performance and fairness [55, 156]. A reasonable question to ask is how these schedulers interact with the page walk scheduler.

Page walk schedulers play an important role in reducing address translation overheads, which none of these other schedulers aim to do. Thus, even in the presence of sophisticated wavefront and memory controller schedulers, we expect that improvements to page walk scheduling will still be useful. The page walk scheduler is unlikely to have significant interactions with the memory schedulers as the maximum amount of memory traffic from the page walk schedulers would still only consume a relatively small fraction of a GPU's total memory bandwidth. That said, there still could be opportunities for better coordination among the different schedulers, but we leave such explorations for future work.

4.7 Related Work

Three research domains are related to this work: TLB management, scheduling in memory controllers, and work scheduling in GPUs.

4.7.1 TLB Management

The emergence of shared virtual memory (SVM) between the CPU and the GPU as a key programmability feature in a heterogeneous system has made an efficient virtual-to-physical address translation for GPUs a necessity. Lowe-Power *et al.* [175] and Pichai *et al.* [172] were among the first to explore designs GPU MMU. Lowe-Power *et al.* demonstrated that coalescer, shared L2 TLB and multiple independent page walkers are essential components of an efficient GPU MMU design. Their design is similar to our baseline configuration. On the other hand, Pichai *et al.* showed the importance of making wavefront (warp) scheduler to be TLB-aware.

More recently, Vesely *et al.* demonstrated on real hardware, that a GPU's translation latencies can be much longer than that of a CPU's and GPU applications with memory access divergence may bottleneck due to address translation overheads [210]. Cong *et al.* proposed TLB hierarchy similar to our baseline but additionally proposed to use a CPU's page table walkers for GPUs [95]. However, accessing CPU page table walkers from a GPU could be infeasible in a real hardware due to longer latencies. Lee *et al.* proposed a software managed virtual memory to provide an illusion of a large memory by partitioning GPU programs to fit into the physical memory space [137]. Ausavarungnirun *et al.* showed that address translation overhead could be even larger in the presence of multiple concurrent applications on a GPU [24]. They selectively bypassed TLBs to avoid thrashing and prioritizing address translation over data access to reduce overheads. Yoon *et al.* demonstrated the significance of address translation overheads in the performance of GPU applications and proposed to employ virtual caches for GPUs to defer address translation only after a cache miss [215].

Different from these works, we demonstrate the importance of *(re-)ordering page table walk requests* and designed a SIMT-aware page table walk scheduler. Most of these works are either already part of our baseline (*e.g.,* [175]) and/or are largely orthogonal to ours (*e.g.,* [24]).

Address translation overheads are well studied in CPUs. To exploit page localities among threads, Bhattacharjee *et al.* proposed inter-core cooperative TLB prefetchers [39]. Pham *et al.* proposed to exploit naturally occurring contiguity to extend effective reach of TLB [169]. Bhattacharjee later proposed shared PWCs and efficient page table designs to increase PWCs hits [33]. Cox *et al.* have proposed `MIX TLBs` that support different page sizes in a single structure [66]. Barr *et al.* proposed `SpecTLB` that speculatively predicts address translations to avoid the TLB miss latency. Several others proposed to leverage segments to selectively bypass TLB and the cost of TLB misses [29, 84, 123]. While some of these techniques can be extended to GPUs, page table walk scheduling is orthogonal to them. Basu *et al.* and Karakostas *et al.* also proposed ways to reduce energy dissipation in a CPU's TLB hierarchy [30, 124].

4.7.2 Scheduling in Memory Controllers

Memory bandwidth has become a potential performance limiter with the emergence of large multi-cores and GPUs [178]. Rixner *et al.* introduced early memory scheduling policies to exploit memory parallelism for better performance [177]. Chatterjee *et al.* proposed staged reads that parallelize read and write requests through two staged read operations and scheduling of writes to take advantage of them [54]. Yoongu *et al.* introduced `ATLAS` that prioritizes threads with least serviced at the memory controller during an epoch [128].

A GPU's SIMT execution exacerbates memory bandwidth bottleneck [45]. Ausavarungnirun *et al.* proposed staged memory scheduling to exploit locality by batching row buffer hit requests [23]. Chatterjee *et al.* proposed a memory scheduler batching requests from the same wavefronts to solve memory access divergence [55]. Our SIMT-aware scheduler bears similarity with this work as we also batch requests but in the context of page walks. Further, Li *et al.* proposed to prioritize memory accesses with higher inter-core locality [138].

The fairness of resource sharing is also important in presence of multiple contenders. Mutlu *et al.* proposed `STFM` that estimates the slowdown of threads due to sharing the DRAM and prioritizes requests from the slowest thread [155]. `PAR-BS` provides QoS by batching and scheduling requests from the same thread [156]. Yoongu *et al.* proposed `TCM` that groups threads with similar memory access patterns and apply different scheduling policies for different groups [129]. Jog *et al.* proposed to allocate fair memory bandwidth among concurrently executing kernels on different CUs in a GPU [118]. Jeong *et al.* proposed a QoS-aware scheduling that prioritizes CPUs with low latency while guaranteeing QoS of GPUs [114]. Usui *et al.* proposed `DASH` that considers the deadline for accelerators instead of always prioritizing CPU workloads [206].

These works focus solely on memory (DRAM), and not on page table walks. However, the existence of such a rich body of work shows the potential of significant follow-on research in exploring various policies for page table walk scheduling for both performance and QoS.

4.7.3 Work Scheduling in GPUs

Smart scheduling of work in the GPU's compute units has been widely investigated, too. Rogers *et al.* proposed `CCWS` that limits the number of active wavefronts on computer units if it detects thrashing on L1 cache [179]. The authors then extended it considering L1 cache usage in wavefront scheduling to reduce the impact of memory access divergence [180]. Li *et al.* extended `CCWS` to also allow bypassing the L1 cache for selected wavefronts when shared resources have additional headroom after limiting wavefronts [139]. Kayrian *et al.* dynamically throttled parallelism in CUs based on application characteristics and contention in the memory subsystem [125]. Unlike these works, we focus on page table walk scheduling. However, an interesting future study could explore interactions between page walk scheduling and scheduling at CUs.

4.8 Conclusion

We demonstrate the importance of reordering page table walk requests for GPUs. The impact of this reordering is particularly severe for irregular GPU applications that suffer from significant address translation overheads. We observed that different SIMD memory instructions executed by a GPU application could require vastly different numbers of memory accesses (*work*) to service their page table walk requests. Our SIMT-aware page table walk scheduler prioritizes page table walks from instructions that require less work to service and further batches page walk requests from the same SIMD instruction to reduce GPU-level stalls. These lead to 30% performance improvement for irregular GPU applications through improved forward progress. While we here proposed a specific SIMT-aware scheduler to demonstrate how better page table walk scheduling is valuable, we believe there exists scope for significant follow-on research on page walk scheduling akin to the rich body of work in memory controller scheduling.

Chapter 5

Secure, Consistent, and High-Performance Memory Snapshotting

5.1 Introduction

The notion of acquiring memory snapshots is one of ubiquitous importance to computer systems. Memory snapshots have been used for tasks such as virtual machine migration and backups [148, 217, 83, 79, 68, 164, 103, 91, 52, 62, 14] as well as forensics [51, 191], which is the subject of this work. In particular, memory snapshot analysis is *the* method of choice used by forensic analyses that determine whether a target machine's operating system (OS) code and data are infected by malicious rootkits [26, 101, 165, 166, 167, 50, 70, 190, 69]. Such forensic methods have seen wide deployment. For example, Komoku [166, 165] (now owned by Microsoft) uses analysis of memory snapshots in its forensic analysis, and runs on over 500 million hosts [21]. Similarly, Google's open source framework, Rekall Forensics [9], is used to monitor its datacenters [153]. Fundamentally, all these techniques depend on secure and fast memory snapshot acquisition. Ideally, a memory snapshot acquisition mechanism should satisfy three properties:

① **Tamper resistance.** The target's OS may be compromised with malware that actively evades detection. The snapshot acquisition mechanism must resist malicious attempts by an infected target OS to tamper with its operation.

② **Snapshot consistency.** A consistent snapshot is one that faithfully mirrors the memory state of the target machine at a given instant in time. Consistency is important for forensic tools that analyze the snapshot. Without consistency, different portions of the snapshot may represent different points in time during the execution of the target, making it difficult to assign semantics to the snapshot.

③ **Performance isolation.** Snapshot acquisition must only minimally impact the performance of other applications that may be executing on the target machine.

The security community has converged on three broad classes of techniques for memory snapshot acquisition, namely *virtualization-based*, *trusted hardware-based* and *external hardware-based* techniques. Unfortunately, none of these solutions achieve all three properties (see Figure 5.1).

With virtualization-based techniques (pioneered by Garfinkel and Rosenblum [87]), the target is a virtual machine (VM) running atop a trusted hypervisor. The hypervisor has the privileges to inspect the memory and CPU state of VMs, and can therefore obtain a snapshot of the target. This approach has the benefit of isolating the target VM from the snapshot acquisition mechanism, which is implemented within the hypervisor. However, virtualization-based techniques:

• *impose a tradeoff between consistency and performance-isolation.* To obtain a consistent snapshot, the hypervisor can pause the target VM, thereby preventing the target from modifying the VM's CPU and memory state during snapshot acquisition. But this consistency comes at the cost of preventing applications within the target from executing during snapshot acquisition, which is disruptive if snapshots are frequently required, *e.g.,* when a cloud provider wants to monitor the health of the cloud platform in a continuous manner. The hypervisor could instead allow the target VM to execute concurrently with memory acquisition, but this compromises snapshot consistency.

• *require a substantial software trusted computing base (TCB).* The entire hypervisor is part of the TCB. Production-quality hypervisors have more than 100K lines of code and a history of bugs [71, 72, 73, 74, 75, 131, 183] that can jeopardize isolation.

• *are not applicable to container-based cloud platforms.* Virtualization-based techniques are applicable only in settings where the target is a VM. This restricts the scope of memory acquisition only to environments where the target satisfies this assumption, *i.e.,* server-class systems and cloud platforms that use virtualization. An increasing number of cloud providers are beginning to deploy lightweight client isolation mechanisms, such as those based on containers (*e.g.,* Docker [2]). Containers provide isolation

Property→ **Method**↓	① *Tamper resistance*	② *Snapshot consistency*	③ *Performance isolation*
Virtualization	✓*	Tradeoff: ② ✓ ⇔ ③ ✗	
Trusted hardware	✓	Tradeoff: ② ✓ ⇔ ③ ✗	
External hardware	✗✓**	✗	✓
SnipSnap	✓	✓	✓

Figure 5.1: Design tradeoffs in snapshot acquisition. *(⋆) Virtualized systems provide tamper-resistance assuming that the hypervisor is trusted; however, attacks on hypervisors violate this assumption [71, 72, 73, 74, 75, 131]. (⋆⋆) While external hardware-based techniques were originally thought to be tamper-resistant, a number of attacks have allowed malicious OSes to evade detection by hiding state from the external hardware mechanism [182, 113, 130].*

by enhancing the OS. On container-based systems, obtaining a full-system snapshot would require trusting the OS, and therefore placing it in the TCB. However, doing so defeats the purpose of snapshot acquisition if the goal is to monitor the OS itself for rootkit infection.

Hardware-based techniques reduce the software TCB and are applicable to any target system that has the necessary hardware support. Methods that use trusted hardware rely on the hardware architecture's ability to isolate the snapshot acquisition system from the rest of the target. For example, ARM TrustZone [15, 199, 25, 89] partitions the processor's execution mode so that the target runs in a deprivileged world ("Normal world"), without access to the snapshot acquisition system, which runs in a privileged world ("Secure world") with full access to the target. However, because the processor can only be in one world at any given time, this system offers the same snapshot consistency versus performance isolation tradeoff as virtualized solutions. The situation is more complicated on a multi-processor TrustZone-based system, because the ARM specification allows individual processor cores to independently transition between the privileged and deprivileged worlds [15, §3.3.5]. Thus, from the perspective of snapshot consistency, care has to be taken to ensure that when snapshot acquisition is in progress on one processor core, all the other cores are paused and do not make concurrent updates to memory. This task is impossible to accomplish without some support from the OS to pause other cores. Trusting the OS to accomplish this task defeats the purpose of snapshot acquisition if the goal is to monitor the OS itself.

External hardware-based techniques use a physically isolated hardware module, such as a PCI-based co-processor (*e.g.,* as used by Komoku [21]), on the target system and perform snapshot acquisition using remote DMA (*e.g.,* [108, 150, 26, 165, 47, 166, 141, 136, 152]). These techniques offer performance-isolation by design—the co-processor executes in parallel with the CPU of the target system and therefore fetches snapshots without pausing the target. However, this very feature also compromises consistency because memory pages in a single snapshot may represent the state of the system at different points in time. Further, a malicious target OS can easily subvert snapshot acquisition despite physical isolation of the co-processor [182]. Co-processors rely on the target OS to set up DMA. On modern chipsets with IOMMUs, a malicious target OS can simply program the IOMMU to reroute DMA requests away from physical memory regions that it wants to hide from the co-processor (*e.g.,* pages that store malicious code and data). Researchers have also discussed address-translation attacks that leverage the inability of co-processors to view the CPU's page-table base register (PTBR) [113, 130]. These attacks enable malicious virtual-to-physical address translations, which effectively hide memory contents in the snapshot from forensic analysis tools.

Contributions. We propose and realize **Secure and Nimble In-Package Snapshotting** or SnipSnap, a hardware-based memory snapshot acquisition mechanism that achieves all three properties. SnipSnap frees snapshotting from the shackles of the consistency-performance tradeoff by leveraging two related hardware trends—the emergence of high-bandwidth DRAM placed on the same package as the CPU [44, 142, 143, 97], and the resurgence of near-memory processing [102, 20, 19]. Specifically, processor vendors have embraced technologies like embedded on-package DRAM in products including IBM's Power 7 processor, Intel's Haswell, Broadwell, and Skylake processors, and even in mobile platforms like Apple's iPhone [3]. More recently, higher bandwidth on-package DRAM has been implemented on Intel's Knight's Landing chip, while emerging 3D and 2.5D die-stacked DRAM is expected to be widely adopted [142]. On-package DRAM has in turn prompted flurry of research on near-memory processing techniques that place logic close to these DRAM technologies. Consequently, near-memory processing logic for machine learning, graph processing, and general-purpose processing has been proposed [102, 20, 19] for better system performance and energy.

SnipSnap leverages these hardware trends to realize fast and effective memory snapshotting. SnipSnap leverages on-package DRAM by realizing a fully hardware-based TCB. With modest hardware modifications that increase chip area by under 1%, SnipSnap captures and digitally signs pages in the on-package DRAM. The resulting snapshot captures the memory and CPU state of the machine faithfully, and any attempts by a malicious target OS to corrupt the state of the snapshot can be detected during snapshot analysis. Because SnipSnap's TCB consists only of the hardware, it can be used on target machines running a variety of software stacks, *e.g.,* traditional systems (OS atop bare-metal), virtualized systems, and container-based systems. We identify *consistency* as an important property of memory snapshots and present SnipSnap's memory controller that offers both consistency and performance isolation. We implement SnipSnap using real-system hardware emulation and detailed software simulation atop state-of-the-art implementations of on-package die-stacked DRAM (*e.g.,* UNISON cache [116]). We vary on-package die-stacked DRAM from 512MB to 8GB capacities. We find that SnipSnap offers 4-25$\times$ performance improvements while also ensuring consistency. Finally, we verify SnipSnap's consistency guarantees using TLA+ [135].

In summary, SnipSnap securely obtains consistent snapshots while offering performance-isolation using non-exotic hardware that is already being implemented by chip vendors. This makes SnipSnap a powerful and general approach for snapshot acquisition, with applications to memory forensics and beyond.

5.2 Overview and Threat Model

SnipSnap allows a forensic analyst to acquire a complete snapshot of a target machine's off-chip DRAM memory. SnipSnap's mechanisms are implemented in a hardware TCB and an untrusted snapshot driver in the target's OS. The hardware TCB consists of on-package DRAM, simple near-memory processing logic, and requires modest modification of the on-chip memory controller and CPU register file. In concert, these components

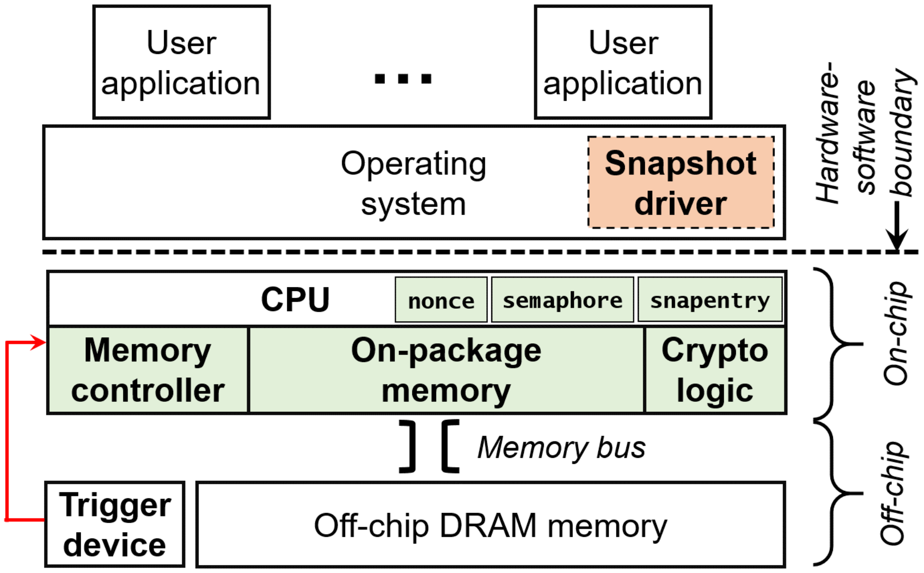


Figure 5.2: Architecture of SnipSnap. Only the on-chip hardware components are in the TCB.

operate as described below.

A forensic analyst initiates snapshot acquisition by triggering the hardware to enter *snapshot mode*. Subsequently, the memory controller iteratively brings each physical page frame from off-chip DRAM to the on-package DRAM. SnipSnap's on-chip near-memory processing logic creates a copy of the page and computes a cryptographic digest of the page. The untrusted snapshot driver in the target OS then commits the snapshot entry to an external medium, such as persistent storage, the network, or a diagnostic serial port. The hardware exits snapshot mode after the near-memory processing logic has iterated over all page frames of the target's off-chip DRAM. A well-formed memory snapshot from SnipSnap contains one snapshot entry per page frame and one entry with CPU register state and a cryptographic digest. Figure 5.2 shows the components of SnipSnap:

① The *trigger device* is an external mechanism that initiates snapshot acquisition. When activated, the trigger device toggles the hardware into snapshot mode. It also informs the target's OS that the hardware has entered snapshot mode.

② The *memory controller* brings pages from off-chip DRAM into on-package DRAM to be copied into the snapshot when the hardware is in snapshot mode (as discussed above). The memory controller maintains internal hardware state to sequentially iterate over all off-chip DRAM page frames. The main novelty in SnipSnap's memory controller is a copy-on-write feature that allows snapshot acquisition to proceed without pausing the target.

③ The *near-memory processing logic* implements cryptographic functionality for hash and digital-signature computation in on-package DRAM [56]. As we show, such near-memory processing is readily implemented atop, for example, die-stacked memory [142]. As such, we assume that the hardware is endowed with a public/private key pair (as are TPMs—trusted platform modules). Digital signatures protect the integrity of the snapshot even from an adversary with complete control of the target's software stack.

④ The *snapshot driver*, SnipSnap's only software component, is implemented within the target's OS. Its sole responsibility is to copy snapshot entries created by the hardware to a suitable external medium.

⑤ The *hardware/software interface* facilitates communication between the snapshot driver and the hardware components. This interface consists of three special-purpose registers and adds minimal overhead to the existing register file of modern processors, which typically consists of several tens of architecturally-visible and hundreds of physical registers.

Threat Model. Our threat model is that of an attacker who controls the target's software stack and tries to subvert snapshot acquisition. The attacker may try to corrupt the snapshot, return stale snapshot entries, or suppress parts of the snapshot. A snapshot produced by SnipSnap must therefore contain sufficient information to allow a forensic analyst to verify integrity, freshness, and completeness of the snapshot. We assume that the on-chip hardware components described above are trusted and are part of the TCB. We exclude physical attacks on off-chip hardware components, *e.g.,* those that modify contents of pages either in off-chip DRAM via electromagnetic methods, or as they transit the memory bus.

SnipSnap's snapshot driver executes within the target OS, which may be controlled by the attacker. We will show that despite this, a corrupt snapshot driver cannot compromise snapshot integrity, freshness, or completeness. At worst, the attacker can misuse his control of the snapshot driver to prevent snapshot entries (or the entire snapshot) from being written out to the external medium. However, the forensic analyst can readily detect such denial of service attacks because the resulting snapshot will be incomplete. Once the forensic analyst obtains a snapshot, he can analyze it using methods described in prior work (*e.g.,* [167, 26, 50, 70, 101, 166, 69, 81]) to determine if the target is infected with malware.

SnipSnap's main goal is secure, consistent, and fast memory snapshot acquisition. Forensic analysts can perform offline analyses on these snapshots, *e.g.,* to check the integrity of the OS kernel or to detect traces of malware activity. While analysts can use SnipSnap to request snapshots for offline analysis as often as they desire, it is not a tool to perform continuous, event-based monitoring of the target machine. To our knowledge, state of the art forensic tools to detect advanced persistent threats (*e.g.,* [26, 101, 165, 166, 167, 50, 70, 190, 69, 21]) rely on offline analysis of memory snapshots.

5.3 Design of SnipSnap

We now present SnipSnap's design, beginning with a discussion of snapshot consistency.

5.3.1 Snapshot Consistency

A snapshot of a target machine is *consistent* if it reflects the state of the target machine's off-chip DRAM memory pages and CPU registers at a given instant in time. Consistency is an important property for forensic applications that analyze snapshots. Without consistency, different memory pages in the snapshot represent the state of the target at different points in time, causing the forensic analysis to be imprecise. For example, consider a forensic analysis that detects rootkits by checking whether kernel data structures satisfy certain invariants, *e.g.,* that function pointers only point to valid function targets [167]. Such forensic analysis operates on the snapshot by identifying pointers in kernel data structures, recursively traversing these pointers to identify more data structures in the snapshot, and checking invariants when it finds function pointers

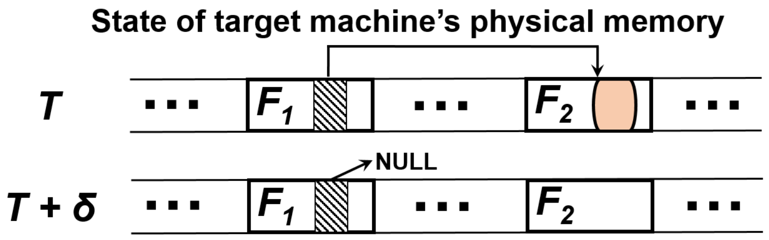


Figure 5.3: Example showing need for snapshot consistency. *Depicted above is the memory state of a target machine at two points in time, T and T+δ. At T, a pointer in F_1 points to an object in F_2. At T+δ, the object has been freed and the pointer set to* NULL. *Without consistency, the snapshot could contain a copy of F_1 at time T and F_2 at time T+δ (or vice-versa), causing problems for forensic analysis.*

in the data structures. If a page F_1 of memory contains a pointer to an object allocated on a page F_2, and the snapshot acquisition system captures F_1 and F_2 in different states of the target, then the forensic analysis can encounter a number of illogical situations (Figure 5.3). Such inconsistencies can also be used to hide malicious code and data modifications in memory [113]. Prior work [167, 26] encountered such situations in the analysis of inconsistent snapshots, and had to resort to unsound heuristics to remedy the problem. A consistent snapshot will capture the state of the target's memory pages at either T or at T+δ, thereby allowing the forensic analysis to traverse data structures in memory without the above problems.

As discussed in Section 5.1, prior systems have achieved snapshot consistency at the cost of performance isolation, or vice versa. SnipSnap acquires consistent memory snapshots without pausing the target machine in the common case. Snapshot acquisition proceeds in parallel with user applications and kernel execution that can actively modify memory. SnipSnap's hardware design ensures that the acquired memory snapshot reflects the state of the target machine at the instant when the hardware entered snapshot mode.

Consistency versus Quiescence. While SnipSnap ensures that an acquired snapshot faithfully mirrors the state of the machine at a given time instant, we do not specify what that time instant should be. Specifically, while snapshot consistency is a *necessary* property for client forensic analysis tools, it is not *sufficient*, *i.e.,* not every consistent snapshot is ideal from the perspective of client forensic analyses. For example, consider a consistent snapshot acquired when the kernel is in the midst of creating a new process. The kernel may have created a structure to represent the new process but may not have finished adding it to the process list, resulting in a snapshot where the process list is not well-formed.

In response, prior work suggests collecting snapshots when the target machine is in *quiescence* [101], *i.e.,* a state of the machine when kernel data structures are likely to be well-formed. Quiescence is a domain-specific property that depends on which data structures are relevant for the forensic analysis and what it means for them to be well-formed. SnipSnap only guarantees consistency, and relies on the forensic analyst to trigger snapshot acquisition at an instant when the system is quiescent. Because SnipSnap guarantees consistency, even if the target enters a non-quiescent state after snapshot acquisition has been triggered, *e.g.,* due to concurrent kernel activity initiated by user applications, the snapshot will reflect state of the target at the beginning of the snapshot acquisition. Triggering snapshot acquisition when the system is in *non-quiescent* state may require a forensic analyst to retake the snapshot.

5.3.2 Triggering Snapshot Acquisition

An analyst requests a snapshot using SnipSnap's trigger device. This device accomplishes three tasks: ① it toggles the hardware TCB into snapshot mode; ② it informs the target's OS that the hardware is in snapshot mode; and ③ it allows the analyst to pass a random nonce that is incorporated into the cryptographic digest of the snapshot.

Task ① requires direct hardware-to-hardware communication between the trigger device and the hardware TCB that is transparent to, and therefore cannot be compromised by, the target OS. Commodity systems offer many options to implement such communication, and SnipSnap can adapt any of them. For example, we could connect a physical device to the programmable interrupt controller, and have it deliver a non-maskable interrupt to the processor when it is activated. Upon receipt of this interrupt, the hardware TCB examines the IRQ to determine its origin, and switches to snapshot mode. Since this triggering mechanism piggybacks on the standard pin-to-bus interface, we find that implementing it requires less than 1% additional area on the hardware TCB.

Another possibility, which allows a forensic analyst to remotely trigger snapshot acquisition, is to use Wakeup-on-LAN (WoL) mechanisms available on modern network interface controllers (NICs). WoL is an Ethernet standard that allows a computer to be turned on by a specially-crafted network message, known as a *magic packet* (*e.g.,* [78]). The idea is to turn on sleeping processors solely with hardware support, without OS involvement. To adapt WoL to SnipSnap, we add a hardware finite state machine (FSM) in the NIC, similar to that required by WoL, to recognize a specially-crafted *snapshot-trigger packet*. When the FSM detects the packet, it signals the ACPI control on the motherboard. Current systems already maintain hardware via the ACPI to transition the cores and memory system from an off (or S5 power) state. SnipSnap can piggyback on these same channels and buses to toggle the hardware into snapshot acquisition mode. We have evaluated the hardware cost of this scheme. Our snapshot-trigger packet FSM hardware requires under a 5% area increase in the NIC logic. The motherboard is left largely unaffected while the extra channels and bus messages used to transition the memory controller require less than a 0.5% area increase in the hardware TCB.

Task ② is to inform the OS, so that it can start executing the snapshot driver. Depending on the implementation of task ①, this task can be accomplished by raising an interrupt or an ACPI event. The target OS invokes the snapshot driver from the interrupt handler or ACPI event handler.

To accomplish task ③, we assume that the trigger device is equipped with device memory that is readable from the target OS. The analyst writes the nonce to device memory, and the OS reads it from there, *e.g.,* after mounting the device as `/dev/trigger_device`.

For example, if the trigger device is implemented using a WoL-like mechanism in the NIC, the analyst could send the nonce in a second packet following the snapshot-trigger packet. The NIC could write the nonce into its device memory after recognizing the snapshot-trigger packet, and the OS could read it from the NIC.

5.3.3 DRAM and Memory Controller Design

SnipSnap relies on on-package DRAM for secure and consistent snapshots. Today, research groups are actively studying how best to organize on-package DRAM. Research questions focus on whether on-package DRAM should be organized as a hardware cache of the off-chip DRAM *i.e.,* the physical address space is equal to the off-chip DRAM capacity [144, 176, 116], or should extend the physical address space instead, *i.e.,* the physical address space is the sum of the off-chip DRAM and on-chip memory capacities [63, 214]. While SnipSnap can be implemented on any of these designs, we focus on die-stacked DRAM caches as they have been widely studied and are expected to represent initial commercial implementations [144, 176, 116, 117].

DRAM caches can be designed in several ways. They can be used to cache data in units of cache lines like conventional L1-LLCs [144, 176, 116]. Unfortunately, the fine granularity of cache lines results in large volumes of tag metadata stored in either SRAM or DRAM caches themselves [144, 176, 116, 117]. Thus, architects generally prefer to organize DRAM caches at page-level granularity. While SnipSnap can be built using any DRAM cache data granularity, we focus on such page-level data caching approaches.

Overall, as a hardware-managed cache, the DRAM cache is not directly addressable from user- or kernel-mode. Further, all DRAM references are mediated by an on-chip memory controller, which is responsible for relaying the access to on-package or off-chip DRAM. That is, CPU memory references are first directed to per-core MMUs before being routed to the memory controller, while device memory references (*e.g.,* using DMA) are directed to the IOMMU before being routed to the memory controller.

Regular Operation. When snapshot acquisition is not in progress, SnipSnap's on-package memory acts as a hardware DRAM cache, before off-chip DRAM (see Figure 5.4(a)). The DRAM cache stores data in the unit of pages, and maintains tags, as is standard, to identify the frame number of the page cached and additional bits to denote usage information, like valid and replacement policy bits. When a new page must be brought into an already-full cache, the memory controller evicts a victim using standard replacement policies.

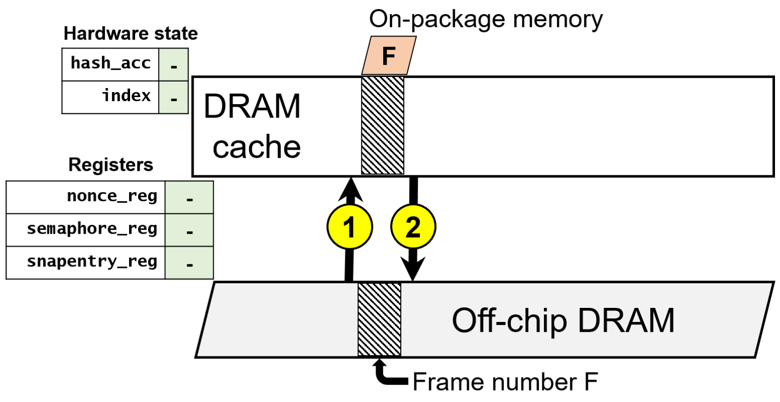


Figure 5.4(a) During regular operation*, on-chip memory is a cache of off-chip DRAM pages.* ① *Accesses by the CPU to a DRAM page brings the page to the on-chip memory, where it is tagged using its frame number (F).* ② *Pages are evicted from on-chip memory region when it reaches its capacity.*

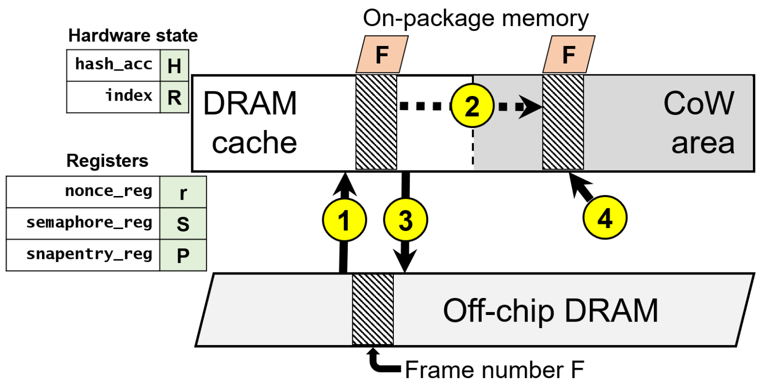


Figure 5.4(b) In snapshot mode*, on-chip memory is split in two.* ① *The DRAM cache works as in Figure 5.4(a).* ② *If there is a write to a page that has not yet been snapshot (i.e., $F \geq R$), it is copied into the CoW area.* ③ *The page may be evicted if the DRAM cache reaches capacity.* ④ *The CoW area copy of the page remains until it has been included in the snapshot (i.e., $F < R$), after which it is overwritten with other pages that enter the CoW area. In snapshot mode H and R are initialized to 0.*

Figure 5.4: Layout of on-chip memory.

Snapshot Mode. When the trigger device signals the hardware to enter snapshot mode, several hardware operations occur. First, the hardware captures the CPU register state of the machine (across all cores). Second, all CPUs are paused, their pipelines are drained, their cache contents flushed (if CPUs use write-back caches), and their load-store queues and write-back buffers drained. These steps ensure that all dirty cache line contents are updated in main memory before snapshot acquisition begins. Third, SnipSnap's memory controller reconfigures the organization of on-package DRAM to ensure that a consistent snapshot of memory is captured. It must track any modifications to memory pages that are not yet included in the snapshot and keep a copy of the original page till it has been copied to the snapshot.

To achieve this goal, the memory controller splits the on-package DRAM into two portions (Figure 5.4(b)). The first portion continues to serve as a cache of off-chip DRAM memory. Since only this portion of on-package DRAM is available for caching in snapshot mode, the memory controller tries to fit in it all the pages that were previously cached during regular operation into the available space. If all pages cannot be cached, the memory controller selects and evicts victims to off-chip DRAM. The second portion of die-stacked memory serves as a copy-on-write (CoW) area. The CoW area allows user applications and the kernel to modify memory concurrently with snapshot acquisition, while saving pages that have not yet been included in the snapshot. We study several ways to partition on-package DRAM into the CoW and DRAM cache areas in Section 5.6.

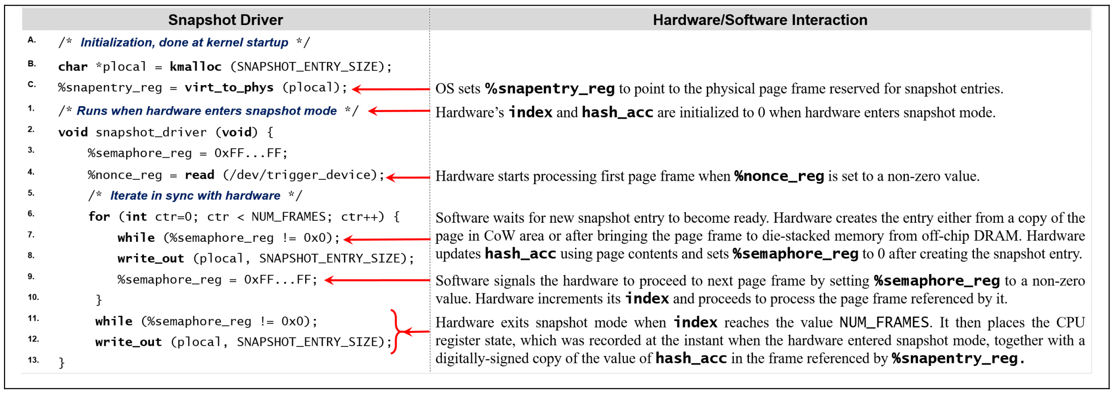
Recall that a snapshot contains a copy of all pages in off-chip DRAM memory. However, the hardware creates a snapshot entry one page of memory at a time. It works in tandem with the snapshot driver to write this snapshot entry to an external medium and then iterates to the next page of memory until all pages are written out to the snapshot. As this iteration proceeds, other applications and the kernel may concurrently modify memory pages that have not yet been included in the snapshot. If SnipSnap's memory controller sees a write to a memory page that the hardware has not yet copied, the memory controller creates a copy of the original page in the CoW area, and lets the write operation proceed in the DRAM cache area. A page frame is copied *at most once* into the CoW area, and this happens only if the page has to be modified by other applications before it has been copied into the snapshot.

The memory controller maintains internal hardware state in the form of an *index* that stores the frame number (R in Figure 5.4(b)) of the page that is currently being processed for inclusion in the snapshot. The hardware initializes the index to 0 when it enters snapshot mode. The memory controller uses the index as follows. It copies a frame F from the DRAM cache to the CoW area when it has to write to that frame and $\mathsf{F} \geq \mathsf{R}$, indicating that the hardware has not yet iterated to frame F to create a snapshot entry for it. If $\mathsf{F} < \mathsf{R}$, then it means that the frame has already been included in the snapshot, and can be modified without copying it to the CoW area. SnipSnap requires that page frames be copied into the snapshot sequentially in ascending order by frame number.

To create a new snapshot entry for a page frame, the memory controller first checks whether this page frame is in the CoW area. If it exists, the hardware proceeds to create a snapshot entry using that copy of the page. The memory controller can then reuse the space occupied by this page in the CoW area. If the page frame is not in the CoW area, the memory controller checks to see if it already exists in the DRAM cache. If not, it brings the page from off-chip DRAM into the DRAM cache, from where the hardware creates a snapshot entry for that page. It places the newly-created entry in a physical page frame referenced by the *snapshot entry register* (`snapentry_reg` in Figure 5.4), and informs the snapshot driver using the *semaphore register* (`semaphore_reg` in Figure 5.4). The driver then writes out the entry to a suitable external medium and informs the hardware, which increments the index and iterates to the next page frame.

The hardware exits snapshot mode when the index has iterated over all the frames of off-chip DRAM. At this point, the hardware creates a snapshot entry containing the CPU register state (captured on entry into snapshot mode), and appends it as the last entry of the snapshot. We leverage die-stacked logic to capture and record register state. SnipSnap's approach is inspired by prior work on introspective die-stacked logic [157], where hardware analysis logic built on die-stacked layers uses probes or "stubs" on the CPUs to introspect on dynamic type analysis, data flight recorders, *etc.* Similarly, we design hardware support to capture register state, using: ① stubs that allow the contents of the register file to be latched into the logic on the die-stack; and ② logic on the die-stack that copies the contents of register files into the last snapshot entry.

The memory controller's use of CoW ensures that concurrent applications can make



Snapshot Driver	Hardware/Software Interaction
`/* Initialization, done at kernel startup */`	
`char *plocal = kmalloc (SNAPSHOT_ENTRY_SIZE);`	
`%snapentry_reg = virt_to_phys (plocal);`	OS sets **%snapentry_reg** to point to the physical page frame reserved for snapshot entries.
`/* Runs when hardware enters snapshot mode */`	Hardware's **index** and **hash_acc** are initialized to 0 when hardware enters snapshot mode.
`void snapshot_driver (void) {`	
`    %semaphore_reg = 0xFF...FF;`	
`    %nonce_reg = read (/dev/trigger_device);`	Hardware starts processing first page frame when **%nonce_reg** is set to a non-zero value.
`    /* Iterate in sync with hardware */`	
`    for (int ctr=0; ctr < NUM_FRAMES; ctr++) {`	
`        while (%semaphore_reg != 0x0);` `        write_out (plocal, SNAPSHOT_ENTRY_SIZE);`	Software waits for new snapshot entry to become ready. Hardware creates the entry either from a copy of the page in CoW area or after bringing the page frame to die-stacked memory from off-chip DRAM. Hardware updates **hash_acc** using page contents and sets **%semaphore_reg** to 0 after creating the snapshot entry.
`        %semaphore_reg = 0xFF...FF;` `    }`	Software signals the hardware to proceed to next page frame by setting **%semaphore_reg** to a non-zero value. Hardware increments its **index** and proceeds to process the page frame referenced by it.
`    while (%semaphore_reg != 0x0);` `    write_out (plocal, SNAPSHOT_ENTRY_SIZE);` `}`	Hardware exits snapshot mode when **index** reaches the value NUM_FRAMES. It then places the CPU register state, which was recorded at the instant when the hardware entered snapshot mode, together with a digitally-signed copy of the value of **hash_acc** in the frame referenced by **%snapentry_reg**.

Figure 5.5: Pseudocode of the snapshot driver and the corresponding hardware/software interaction.

progress, while still maintaining the original copies of memory pages for a consistent snapshot. The hardware pauses a user application during snapshot acquisition only when the CoW area fills to capacity and when that application attempts to write to a page that the hardware has not yet included in the snapshot. In this case, the hardware can resume these applications when space is available in the CoW area, *i.e.,* when a page from there is copied to the snapshot.

Our implementation of SnipSnap has important design implications on recently-proposed DRAM caches. Research has shown that DRAM caches generally perform most efficiently when they use page-sized allocation units to reduce tag array size requirements [117, 116]. However, they also employ memory usage predictors (*e.g.,* footprint predictors [117, 116]) to fetch only the relevant 64B blocks from a page, thereby efficiently using scarce off-chip bandwidth by not fetching blocks that will not be used. This means the following for SnipSnap. During regular operation, SnipSnap continues to employ page-based DRAM caches with standard footprint prediction. However, to simplify our design, SnipSnap does not use footprint prediction during snapshot mode and moves entire pages of data with their constituent cache lines in both the CoW and DRAM cache partitions. Naturally, this does degrade performance of applications running simultaneously with snapshotting; however, our results (see Section 5.6) show that performance improvements versus current snapshotting techniques remain high.

5.3.4 Near-Memory Processing Logic

Near-memory processing logic implements cryptographic functionality to create the snapshot. On a target machine with N frames of off-chip DRAM memory, the snapshot itself contains $\mathsf{N}+1$ entries. The first N entries store, in order, the contents of page frames 0 to N-1 of memory (thus, an individual snapshot entry is 4KB). The last entry of the snapshot stores the CPU register state and a cryptographic digest that allows a forensic analyst to determine the integrity, freshness and completeness of the snapshot.

The near-memory processing logic maintains an internal *hash accumulator* that is initialized to zero when the hardware enters snapshot mode. It updates the hash accumulator as the memory controller iterates over memory pages, recording them in the snapshot. Suppose that we denote the value of the hash accumulator using $\mathsf{H}_{\mathsf{idx}}$, where idx denotes the current value of the memory controller's index (thus, $\mathsf{H}_0 = 0$). When the memory controller creates a snapshot entry for page frame numbered idx, the near-memory processing logic updates the value of the hash accumulator to $\mathsf{H}_{\mathsf{idx}+1}=\mathsf{Hash}(\mathsf{idx} \parallel \mathsf{r} \parallel \mathsf{H}_{\mathsf{idx}} \parallel \mathsf{C}_{\mathsf{idx}})$.

Here:

① The value idx is the hardware's index. It records the frame number of the page that included in the snapshot;

② The value r denotes a random nonce supplied by the forensic analyst using the trigger device and stored in the on-chip *nonce register* (`nonce_reg` in Figure 5.4(b)). The use of the nonce ensures freshness of the snapshot;

③ $\mathsf{H}_{\mathsf{idx}}$ denotes the current value of the hash accumulator;

④ $\mathsf{C}_{\mathsf{idx}}$ denotes the actual contents of page frame idx.

All these values are readily available on-chip.

When the memory controller finishes iterating over all N memory page frames, the value H_{N} in the hash accumulator in effect denotes the value of a *hash chain* computed cumulatively over all off-chip DRAM memory pages. The final snapshot entry enlists the values of CPU registers as recorded by the hardware when it entered snapshot mode—let us denote the CPU register state using $\mathsf{C}_{\mathsf{reg}}$. The near-memory logic updates the hash accumulator one final time to create $\mathsf{H}_{\mathsf{N+1}}=\mathsf{Hash}(\mathsf{N} \parallel \mathsf{r} \parallel \mathsf{H}_{\mathsf{N}} \parallel \mathsf{C}_{\mathsf{reg}})$. It digitally signs $\mathsf{H}_{\mathsf{N+1}}$ using the hardware's private key, and records the digital signature in the last entry of the snapshot. This digital signature assists with the verification of snapshot integrity (Section 5.4). We use SHA-256 as our hash function, which outputs a 32-byte hash value. The size of the digital signature depends on the key length used by the hardware. For instance, a 1024-bit RSA key would produce a 86-byte signature for a 32-byte hash value with OAEP padding.

5.3.5 Snapshot Driver and HW/SW Interface

The hardware relies on the target's OS to externalize the snapshot entries that it creates. We rely on software support for this task because it simplifies hardware design, and also provides the forensic analyst with considerable flexibility in choosing the external medium to which the snapshot must be committed. Although we rely on the target OS for this critical task, we do not need to trust the OS and even a malicious OS cannot corrupt the snapshot created by the hardware.

The hardware and the software interact via an interface consisting of three registers (nonce, snapshot entry and semaphore registers), which were referenced earlier. Figure 5.5 shows the software component of SnipSnap and the hardware/software interaction. SnipSnap's software component consists of initialization code that executes at kernel startup (lines A–C) and a snapshot driver that is invoked when the hardware enters snapshot mode (lines 1–13). The implementation of the snapshot driver in the target OS depends on the trigger device and executes as a kernel thread. For example, if the trigger device raises an interrupt to notify the target OS that the hardware has switched to snapshot mode, the snapshot driver can be implemented within the corresponding interrupt handler. If the trigger device instead uses ACPI events for notification, the snapshot driver can be implemented as an ACPI event handler.

In the initialization code, SnipSnap allocates a buffer (the `plocal` buffer) that is the size of one snapshot entry. This buffer serves as the temporary storage area in which the hardware stores entries of the snapshot before they are committed to an external medium. It then obtains and stores the physical address translation of `plocal` in `snapentry_reg`, The hardware uses this physical address to store computed snapshot entries into the `plocal` buffer and the snapshot driver writes it out. Pages allocated using `kmalloc` cannot be moved, ensuring that the buffer is in the same location for the duration of the snapshot driver's execution. If the page moves, *e.g.,* because of a malicious implementation of `kmalloc`, or if `virt_to_phys` returns an incorrect virtual to physical translation, the snapshot will appear corrupted to the forensic analyst.

When hardware enters snapshot mode, it initializes its internal index and hash accumulator, captures CPU register state, and invokes SnipSnap's snapshot driver. The goal of the snapshot driver is to work in tandem with the hardware to create and externalize one snapshot entry at a time. The snapshot driver and the hardware coordinate using the semaphore register, which the driver first initializes to a non-zero value on line 3. It then reads the nonce value that the forensic analyst supplies via the trigger device. Writing this non-zero value into `nonce_reg` on line 4 activates the near-memory processing logic, which creates a snapshot entry for the page frame referenced by the hardware's internal index.

In the loop on lines 6–10, the snapshot driver iterates over all page frames in tandem with the hardware. Each iteration of the loop body processes one page frame. The hardware begins processing the first page of DRAM as soon as line 4 sets `nonce_reg`, and stores the snapshot entry for this page in the `plocal` buffer. On line 7, the driver waits for the hardware to complete this operation. The hardware informs the driver that the `plocal` buffer is ready with data by setting `semaphore_reg` to 0. The driver then commits the contents of this buffer to an external medium, denoted using `write_out` on line 8. The driver then sets `semaphore_reg` to a non-zero value on line 9, indicating to the hardware that it can increment its index and iterate to the next page for snapshot entry creation. Note that the time taken to execute this loop depends on the number of page frames in off-chip DRAM and the speed of the external storage medium.

When the loop completes execution, the hardware would have iterated through all DRAM page frames and exited snapshot mode. When it exits, it writes out the CPU register state captured during snapshot mode-entry and the digitally-signed value of the hash accumulator to the `plocal` buffer, which the snapshot driver can then output on line 12.

5.3.6 Formal Verification

We used TLA+ [135] to formally verify that SnipSnap produces consistent snapshots. To do so, we created a system model that mimics SnipSnap's memory controller in snapshot mode and during regular operation. Our TLA+ system model can be instantiated for various configurations, such as memory sizes, cache sizes, and cache associativities. The model consists of main memory connected to a set of processors. Processors issue read and write requests to the memory hierarchy, each processor can have one outstanding request at a time. Each request is first looked up in the cache, which mimics on-package memory. When it is found in the cache, the request is serviced and the processor is notified to enable it to make more requests. Otherwise, when the request misses the cache, it is sent to memory (mimicing off-chip DRAM) and the cache is populated with the new entry.

When the model switches to snapshot mode, the way in which memory requests are serviced changes. The cache is now split into two with one servicing as the CoW area and the other serving as a reduced-sized cache. A read request operates in the same fashion as during regular operation. A write request requires more care. The address of the request is first looked up in the CoW area. If it is not found there, the model creates a copy of the data (prior to the write) and stores it in the CoW area. The write operation then progresses as before. If the address is found in the CoW area, it means that a prior request to that address has already been made and the data is available in the CoW for the snapshot routine to use it. This enables the write operation to go to the cache.

The switch from regular operation to snapshot mode happens in two steps in our TLA+ model. The model is triggered to switch at any time, and as expected there are many in-flight memory requests being serviced. New requests are stalled, caches are flushed and all queues are emptied before the system switches to snapshot mode and processors resume making memory requests. Once in snapshot mode, all the memory is copied into the snapshot.

We encoded consistency as a safety property by checking that the state of the on-package memory and off-chip DRAM at the instant when the system switches to snapshot mode will be recorded in the snapshot at the end of acquisition. We verified that our system model satisfies this property using the TLA+ model checker. Our TLA+ model of SnipSnap is open source, listed in Appendix A.

5.4 Security Analysis

When a forensic analyst receives a snapshot acquired by SnipSnap, he establishes its integrity, freshness, and completeness. In this section, we describe how these properties can be established, and show how SnipSnap is robust to attempts by a malicious target OS to subvert them.

① Integrity. An infected target OS may attempt to corrupt snapshot entries to hide traces of malicious activity from the forensic analyst. To ensure that the integrity of the snapshot has not been corrupted, an analyst can check the digital signature of the hash accumulator stored in the last snapshot entry. The analyst performs this check by essentially mimicking the operation of SnipSnap's memory controller and near-memory processing logic, *i.e.,* iterating over the snapshot entries in order to recreate the value of the hash accumulator, and verify its digital signature using the hardware's public key. Since the hash accumulator is stored and updated by the hardware TCB, which also computes its digital signature, a malicious target cannot change snapshot entries after they have been computed by the hardware.

② Freshness. The forensic analyst supplies a random nonce via the trigger device when he requests a snapshot. SnipSnap's hardware TCB incorporates this nonce into the hash accumulator computation for each memory page frame, thereby ensuring freshness. Note that SnipSnap uses the untrusted snapshot driver to transfer the nonce from trigger device memory into the hardware's nonce register (line 4 of Figure 5.5). A malicious target OS cannot cheat in this step, because the nonce is incorporated into the hardware TCB's computation of the hash accumulator.

③ Completeness. The snapshot should contain one entry for each page frame in off-chip DRAM and one additional entry storing CPU register state. This criterion ensures that a malicious target OS cannot suppress memory pages from being included in the snapshot. Each snapshot entry is created by the hardware, by directly reading the frame number and page contents from die-stacked memory, thereby ensuring that these entities are correctly recorded in the entry.

Our attack analysis focuses on how a malicious target OS can subvert snapshot acquisition. A forensic analyst uses the trigger device to initiate snapshot acquisition by toggling the hardware TCB into snapshot mode. The trigger device communicates directly with SnipSnap's hardware TCB using hardware-to-hardware communication, transparent to the target's OS, and therefore cannot be subverted by a malicious OS. The hardware then notifies the OS that it is in snapshot mode, expecting the snapshot driver to be invoked.

A malicious target OS may attempt to "clean up" traces of infection before it jumps to the snapshot driver's code so that the resulting snapshot appears clean during forensic analysis. However, once the hardware is in snapshot mode, SnipSnap's memory controller, which mediates all writes to DRAM, uses the CoW area to track modifications to memory pages. Even if the target's OS attempts to overwrite the contents of a malicious page, the original contents of the page are saved in the CoW area to be included in the snapshot. Thus, any attempts by the target OS to hide its malicious activities after the hardware enters snapshot mode are futile. Of course, the target OS could refuse to execute the snapshot driver, which will prevent the snapshot from being written out to an external medium. Such a denial of service attack is therefore readily detectable.

A malicious OS may try to interfere with the execution of the initialization code in lines A–C of Figure 5.5. The initialization code relies on the correct operation of `kmalloc` and `virt_to_phys`. However, we do not have to trust these functions. If `kmalloc` fails to allocate a page, snapshots cannot be obtained from the target, resulting in a detectable denial of service attack. If the pages allocated by `kmalloc` are remapped during execution or `virt_to_phys` does not provide the correct virtual to physical mapping for the allocated space, the `write_out` operation on line 8 will write out incorrect entries that fail the INTEGRITY check.

Once the snapshot driver starts execution, a malicious target OS can attempt to interfere with its execution. If it copies a stale or incorrect value of the nonce into `nonce_reg` from trigger device memory on line 4, the snapshot will violate the FRESHNESS criterion. It could attempt to bypass or short-circuit the execution of the loop on lines 5–10. The purpose of the loop is to synchronize the operation of the snapshot driver with the internal index maintained by SnipSnap's memory controller. If the OS short-circuits the loop or elides the `write_out` on line 8 for certain pages, the resulting snapshot will be missing entries, thereby violating COMPLETENESS. Attempts by the target OS to modify the virtual address of `plocal` or the value of `snapshot_reg` during the execution of the snapshot driver will trigger a violation of INTEGRITY for the same reasons that attacks on the initialization code triggers an INTEGRITY violation.

Finally, a malicious target could try to hide traces of infection by creating a synthetic snapshot that glues together individual entries (with benign content in their memory pages) from snapshots collected at different times. However, such a synthetic snapshot will fail the INTEGRITY check since the hash chain computed over such entries will not match the digitally-signed value in the last snapshot entry.

The last entry records the values of all CPU registers at the instant when the hardware entered snapshot mode. For forensic analysis, the most useful value in this record is that of the page-table base register (PTBR). As previously discussed, forensic analysis of the snapshot often involves recursive traversal of pointer values that appear in memory pages [167, 26, 50, 70, 166, 165, 190]. These pointers are virtual addresses but the snapshot contains physical page frames. Thus, the forensic analysis translates pointers into physical addresses by consulting the page table, which it locates in the snapshot using the PTBR. External hardware-based systems [26, 165, 47, 166, 141, 136, 152] cannot view the processor's CPU registers. Therefore, they depend on the untrusted target OS to report the value of the PTBR. Unfortunately, this results in address-translation redirection attacks [113, 130]. The target OS can create a synthetic page table that contains fraudulent virtual-to-physical mappings and return a PTBR referencing this page table. The synthetic page table exists for the sole purpose of defeating forensic analysis by making malicious content unreachable via page-table translations—it is not used by the target OS during execution. SnipSnap can observe and record CPU register state accurately when the hardware enters snapshot mode and is not vulnerable to such attacks. It captures the PTBR pointing to the page table that is in use when the hardware enters snapshot mode.

5.5 Experimental Methodology

5.5.1 Evaluation Infrastructure

We use a two-step approach to quantify SnipSnap's benefits. In the first step, we perform evaluations on long-running applications with full-system and OS effects. Since this is infeasible with software simulation, we develop hardware emulation infrastructure similar to recent work [160] to achieve this. This infrastructure takes an existing hardware platform, and through memory contention, creates two different speeds of DRAM. Specifically, we use a two-socket Xeon E5-2450 processor, with a total of 32GB of memory, running Debian-sid with Linux kernel 4.4.0. There are 8 cores per socket, each two-way hyperthreaded, for a total of 16 logical cores per socket. Each socket has two DDR3 DRAM memory channels. To emulate our DRAM cache, we dedicate the first socket for execution of our user applications, our kernel-level snapshot driver, and our user-level snapshot process. This first socket hosts our "fast" or on-package memory. The second socket hosts our "slow" or off-chip DRAM. The cores on the second socket are used to create memory contention (using the memory contention benchmark `memhog`, like prior work [169, 170]) such that the emulated die-stacked memory or DRAM cache is 4.5× faster compared to the emulated off-chip DRAM. This provides a similar memory bandwidth performance ratio of a 51.2GBps off-chip memory system compared to a 256GBps of die-stacked memory, consistent with the expected performance ratios of real-world die-stacking [160, 144]. We modify Linux kernel to page between the emulated fast and slow memory, using the `libnuma` patches. We model the timing aspects of paging to faithfully reproduce the performance that SnipSnap's memory controller would sustain. Since our setup models CPUs with write-back caches, we

① **Canneal**	Simulated annealing from PARSEC [40]
② **Dedup**	Storage deduplication from PARSEC [40]
③ **Memcached**	In-memory key-value store [8]
④ **Graph500**	Graph-processing benchmark [5]
⑤ **Mcf**	Memory-intensive benchmark/SPEC 2006 [11]
⑥ **Cifar10**	Image recognition from TensorFlow [12]
⑦ **Mnist**	Computer vision from TensorFlow [12]

Figure 5.6: Description of benchmark user applications.

include the latencies necessary for cache, load-store queue, and write buffer flushes on snapshot acquisition. Finally, we emulate the overhead of marshaling to external media by introducing artificial delays. We vary delay based on several emulated external media, from fast network connections to slower SSDs.

While our emulator includes full-system effects and full benchmark runs, it precludes us from modeling SnipSnap's effectiveness atop recently-proposed (and hence not available commercially) DRAM cache designs. Therefore, we also perform careful software simulation of the state-of-art UNISON DRAM cache [116], building SnipSnap atop it. Like the original UNISON cache paper, we assume a 4-way set-associative DRAM cache with 4KB pages, a 144KB footprint history table, and an accurate way predictor. Like recent work [214], we use an in-house simulator and drive it with 50 billion memory reference traces collected on a real system. We model a 16-core CMP and with ARM A15-style out-of-order CPUs, 32KB private L1 caches, and 16MB shared L2 cache. We study die-stacked DRAM with 4 channels, and 8 banks/rank with 16KB row buffers, and 128-bit bus width, like prior work [117]. Further, we model 16-64GB off-chip DRAM, with 8 banks/rank and 16KB row buffers. Finally, we use the same DRAM timing parameters as as the original UNISON cache paper [116].

5.5.2 Workloads

We study the performance implications of SnipSnap by quantifying snapshot overheads on several memory-intensive applications. We evaluate such workloads since these are the likeliest to face performance degradation due to snapshot acquisition. Even in this "worst-case," we show SnipSnap does not excessively hurt performance.

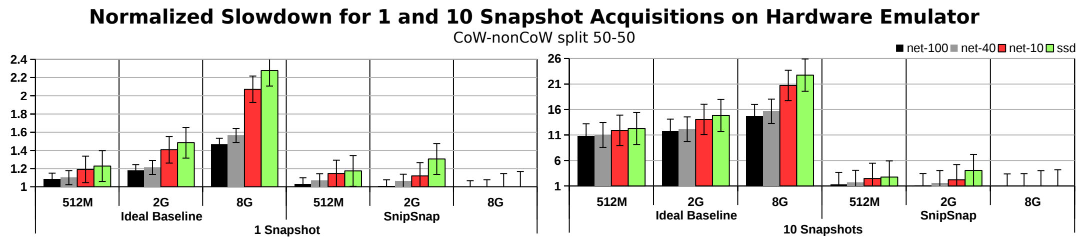


Figure 5.7: Performance impact of snapshot acquisition from hardware emulator studies. *Slowdown caused by modern snapshot mechanisms that also assure consistency, and compare against SnipSnap. We plot results for 1 and 10 snapshots separately (note the different y axes), showing averages, minima, and maxima amongst benchmark runtimes. X-axis shows the amount of on-package memory available on the emulated system. SnipSnap provides 1.2-22× performance improvements against current approaches.*

Figure 5.6 shows our single- and multi-threaded workloads. All benchmarks are configured to have memory footprints in the range of 12-14GB, which exceeds the maximum size of die-stacked memory we emulate (8GB). To achieve large memory footprints, we upgrade the inputs for some workloads with smaller defaults (*e.g.,* Canneal, Dedup, and Mcf), so that their memory usage increases. We set up memcached with a snapshot of articles from the entire Wikipedia database, with over 10 million entries. Articles are roughly 2.8KB on average, but also exhibit high object size variance.

5.6 Evaluation

We now evaluate the benefits of SnipSnap. We first quantify performance, and then discuss its hardware overheads.

5.6.1 Performance Impact on Target Applications

A drawback of current snapshotting mechanisms is that they *must* pause the execution of applications executing on the target to ensure consistency. SnipSnap does not suffer from this drawback. Figures 5.7 and 5.8 quantify these benefits. We plot the slowdown in runtime (lower is better) with benchmark averages, minima, and maxima, as we vary on-package DRAM capacity. We separate performance based on how we externalize snapshots: NICs with 100Gbps, 40Gbps, and 10Gbps throughput, and a solid-state storage disk (SSD) with sequential write throughput of 900MBps. Larger on-package

DRAM (and hence, larger CoW areas) offer more room to store pages that have not yet been included in the snapshot. Faster methods to externalize snapshot entries allow the CoW area to drain quicker. Some of the configuration points that we discuss are not yet in wide commercial use. For example, the AMD Radeon R9, a high-end chipset series supports only up to 4GB of on-package DRAM. Similarly, 40Gbps and 100Gbps NICs are expensive and not yet in wide use.

Figure 5.7 shows results collected on our hardware emulator, assuming that 50% of on-package DRAM is devoted to the CoW area during snapshot mode. We vary the size on-package DRAM from 512MB to 8GB, and assume 16GB off-chip DRAM. Further, our hardware emulator assumes that on-package DRAM is implemented as a page-level fully-associative cache. We show the performance slowdown due to idealized current snapshotting mechanisms, as we take 1 and 10 snapshots. By idealized, we mean approaches like virtualization-based or TrustZone-style snapshotting which require pausing applications on the target to achieve consistency, but which assume unrealizable zero-overhead transition times to TrustZone mode or zero-overhead virtualization. Despite idealization, current approaches perform poorly. Even with only one snapshot, runtime increaseas by 1.2-2.4$\times$ using SSDs. SnipSnap fares much better, outperforming the idealized baseline by 1.2-2.2$\times$, depending on the externalization medium and on-package DRAM size. Snapshotting more frequently (*i.e.,* 10 snapshots) further improves performance by 10.5-22$\times$. Naturally, the more frequent the snapshotting, the more SnipSnap's benefits, though our benefits are significant even with a single snapshot.

Similarly, Figure 5.8 quantifies SnipSnap's performance improvements versus current snapshotting, assuming a baseline with state-of-the-art UNISON cache implementations of on-package DRAM [116], as UNISON cache sizes are varied from 512MB to 8GB. Some key differences between UNISON cache and our fully-associative hardware emulated DRAM cache is that UNISON cache also predicts 64B blocks within pages that should be moved on a DRAM cache miss, and also is implemented as 4-way set associative (as per the original paper). Nevertheless, Figure 5.8 (collected assuming SSDs as the externalizing medium) shows that SnipSnap outperforms idealized versions of

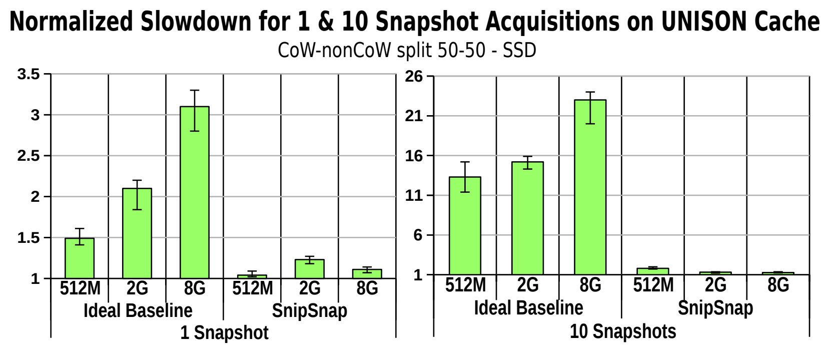


Figure 5.8: Performance impact of snapshot acquisition from simulator studies with UNISON cache [116]. *SnipSnap outperforms idealized versions of current snapshotting approaches by as much as 22× (graphs show benchmark averages, maxima, and minima).*

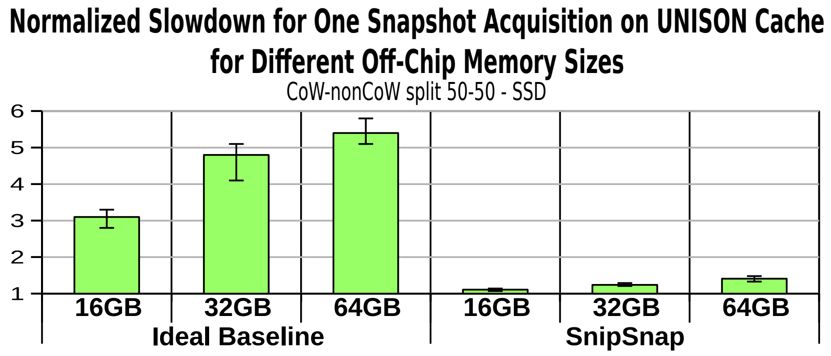


Figure 5.9: Average performance with varying off-chip DRAM size. *Bigger off-chip DRAM takes longer to snapshot, so SnipSnap becomes even more advantageous over current idealized approaches. These results assume UNISON cache with 8GB, split 50:50 in CoW:non-CoW mode during snapshot acquisition and SSDs, taking just one snapshot.*

current snapshotting mechanisms by as much as 22×, and by as much as 3× when just a single snapshot is taken.

SnipSnap's performance also scales far better than idealized versions of current snapshotting with increasing off-chip DRAM capacities. Figure 5.9 compares the performance slowdown due to one snapshot, as off-chip DRAM varies from 16GB to 64GB. These results are collected using UNISON cache (8GB in normal operation, 4GB in snapshot mode, with 4GB CoW), and assuming SSDs. Consider idealized versions of current snapshotting approaches – runtime increases from 3× with 16GB off-chip DRAM to as high as 5.3× with 64GB of memory, when taking just a single snapshot.

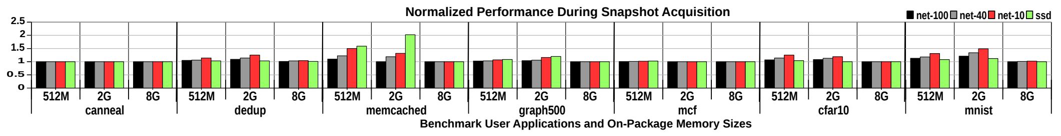


Figure 5.10: Performance impact of snapshot acquisition. *This chart reports the observed performance of user applications executing on the target during snapshot acquisition, normalized against their observed performance during regular execution, i.e., no snapshot acquisition. For each of the seven benchmarks, we report the performance for various sizes of die-stacked memory (50% of which is the CoW area), and for different methods via which the* `write_out` *in Figure 5.5 writes out the snapshot.*

More snapshots further exacerbate this slowdown. While SnipSnap also suffers slowdown with larger off-chip DRAM, it still vastly outperforms current approaches by as much as 5× at 64GB of off-chip DRAM.

So far, we have shown application slowdown comparisons of SnipSnap versus current approaches. Figure 5.10 focuses, instead, on per-benchmark runtime slowdown using SnipSnap, when varying the size of on-package DRAM and the externalizing medium. Results show that most benchmarks, despite being data-intensive, remain unaffected by SnipSnap's snapshot acquisition. The primary exceptions to this are memcached, cfar, and mnist, though their slowdowns vastly outperform current approaches (see Figures 5.7 and 5.8).

5.6.2 CoW Analysis

As discussed in Section 5.3, benchmark runtime suffers during snapshot acquisition only if the CoW area fills to capacity. When this happens, the benchmark stalls until some pages from the CoW area are copied to the snapshot. Figure 5.11 illustrates this fact, and explains the performance of memcached. Figure 5.11 shows the fraction of the CoW area utilized over time during the execution of memcached. The fraction of time for which the CoW area is at 100% directly corresponds to the observed performance of memcached. When CoW utilization is below 100%, as is the case in Figure 5.11(b) the performance of memcached is unaffected.

Next, Figure 5.12 quantifies the performance impact of varying the percentage of die-stacked memory devoted to the CoW area. We vary the split from 50-50% to 25-75% and 75-25% for CoW-nonCoW portions, for various externalization techniques. We

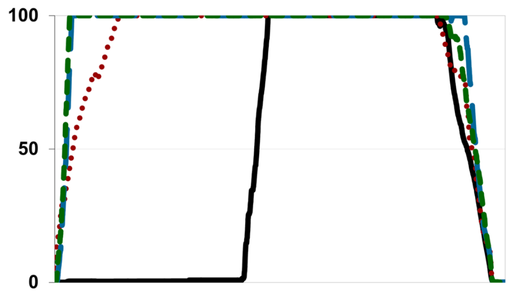


Figure 5.11(a) *512MB of on-chip memory*

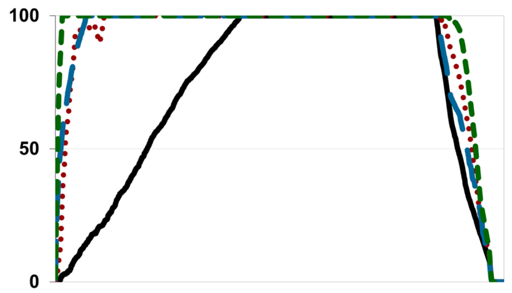


Figure 5.11(b) *1GB of die-stacked memory*

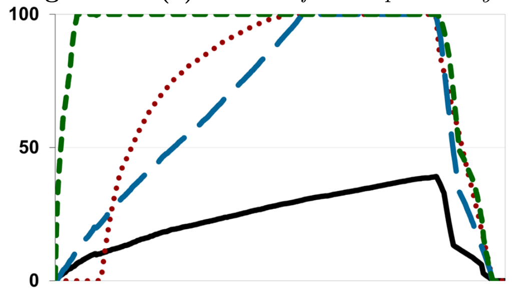


Figure 5.11(c) *2GB of die-stacked memory*

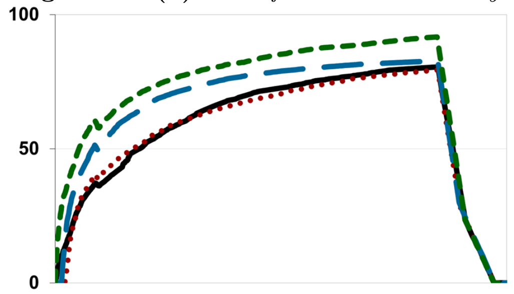


Figure 5.11(d) *4GB of on-chip memory*

Figure 5.11: CoW area utilization over time for memcached. *Y-axis shows CoW area percentage used to store page frames that have not yet been included in the snapshot. X-axis denotes execution progress. We measured CoW utilization for every 1024 snapshot entries recorded. The two charts show CoW utilization trends for various sizes of die-stacked memory and for different methods to write out the snapshot:* ━ **net-100** ••• **net-40** ━ **net-10** ▬ ▬ **ssd**. *Snapshot acquisition does not impact memcached performance when CoW utilization is below 100%.*

present the average results across all workloads for various total die-stacked memory sizes (individual benchmarks follow these average trends). Figure 5.12 shows that performance remains strong across all configurations, even when the percentage of DRAM cache devoted to CoW is low, which potentially leads to more stalls in the system. Furthermore, low CoW only degrades performance at smaller DRAM cache sizes of 512MB, which are smaller than DRAM cache sizes expected in upcoming systems.

Finally, note that the set-associativity of the DRAM cache devoted to the CoW region influences SnipSnap's performance. Specifically, consider designs like UNISON cache [116] (and prior work like Footprint cache [117]), which use 4-way set-associative (and 32-way set-associative) page-based DRAM caches. In these situations, if an entire set of the DRAM cache becomes full (even if other sets are not), applications executing on the target must pause until pages from that set are written to the external medium (*i.e.,* SSD, network, etc.). Even in the worst case (all the application's data maps to a

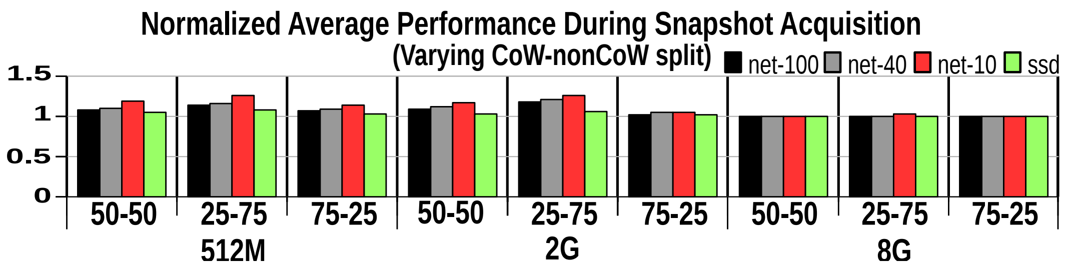


Figure 5.12: Performance impact of snapshot acquisition for different CoW-Cache partitions. *Y-axis shows average performance impact of all benchmarks to take a snapshot, varying CoW-nonCoW partition for different cache sizes. X-axis shows different total sizes of die-stacked memory and various ways in which to partition die-stacked memory for CoW (50%, 25% and 75% for CoW).*

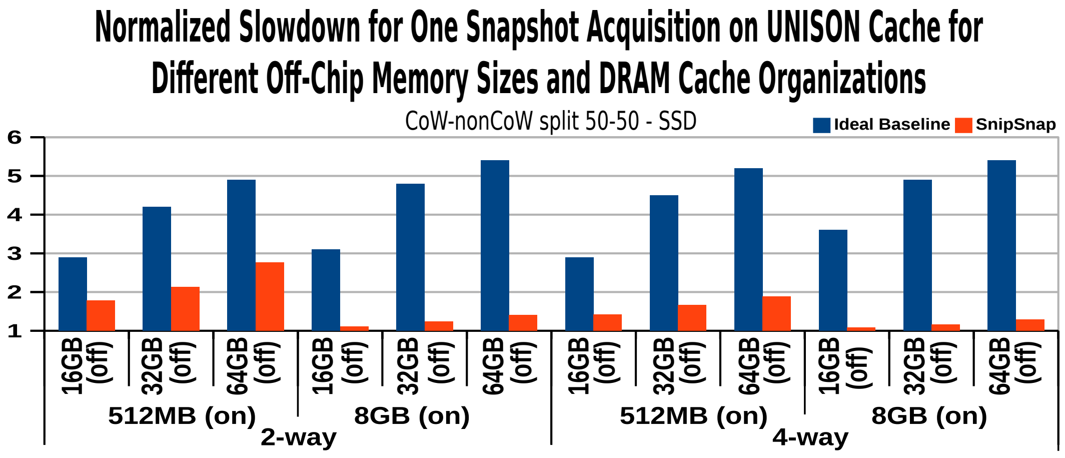


Figure 5.13: Performance as size and set-associativty of UNISON cache changes. *Lower UNISON cache size and set-associativity increases the chances that a set in the CoW region fills up and pauses execution of applications on the target. Results are shown using SSDs, varying off-chip DRAM capacity from 16GB to 64GB, UNISON cache size from 512MB to 8GB, and set-associativity from 2 to 4 way.*

single set so the CoW region always stalls application execution and writing pages to the external medium takes as long as the entire snapshot time) this is *no worse* that idealized versions of current approaches. However, we find that this scenario does not occur in practice. Figure 5.13 quantifies SnipSnap's performance versus an ideal baseline for one snapshot, as off-chip DRAM capacity is varied from 16GB to 64GB, on-chip DRAM capacity is varied from 512MB to 8GB, and associativity is varied between 2-way and 4-way. Larger DRAM caches and higher associativity improve SnipSnap's performance, but even when we hamper UNISON cache to be 512MB and 2-way set-associative, it outperforms idealized current approaches by ~2×. More frequent snapshots further increase this number.

Beyond these studies, we also considered quantifying SnipSnap's performance on a direct-mapped UNISON cache. However, as pointed out by prior work, the conflict misses induced by direct-mapping in baseline designs without snapshotting are so high, that no practical page-based DRAM cache design is direct-mapped [116, 117]. Therefore, we begin our analysis with 2-way set-associative DRAM caches, showing that SnipSnap consistently outperforms alternatives.

5.7 Related Work

As Section 5.1 discusses, there is much prior work on remote memory acquisition based on virtualization, trusted hardware and external hardware. Figure 5.1 characterizes the difference between SnipSnap and this prior work. Aside from these, there are other mechanisms to fetch memory snapshots for the purpose of debugging (*e.g.,* [99, 200, 90, 198, 119]). Because their focus isn't forensic analysis, these systems do not assume an adversarial target OS.

Prior work has leveraged die-stacking to implement myriad security features such as monitoring program execution, access control and cryptography [157, 208, 209, 149, 207, 104, 106, 105]. This work observes that die-stacking allows processor vendors to decouple core CPU logic from "add-ons," such as security, thereby improving their chances of deployment. Our work also leverages additional circuitry on the die-stack to implement the logic needed for memory acquisition. Unlike prior work, which focused solely on additional processing logic integrated using die-stacking, our focus is also on die-stacked memory, which is beginning to see deployment in commercial processors. While SnipSnap also uses the die-stack to integrate additional cryptographic logic and modify the memory controller, it does so to enable near-data processing on the contents of die-stacked memory.

Prior work has also used die-stacked manufacturing technology to detect malicious logic inserted into the processor. The threat model is that of an outsourced chip manufacturer who can insert Trojan-horse logic into the hardware. This work suggests various methods to combat this threat using die-stacked manufacturing. For example, one method is to divide the implementation of a circuit across multiple layers in the stack, each manufactured by a separate agent, thereby obfuscating the functionality of individual layers [107, 204]. Another method is to add logic into die-stacked layers to monitor the execution of the processor for maliciously-inserted logic [43, 42, 41].

There is prior work on near-data processing to enable security applications [93] and modifying memory controllers to implement a variety of security features [194, 212]. There is also work on using programmable DRAM [141] to monitor systems for OS and hypervisor integrity violations. Unlike SnipSnap, which focuses on fetching a complete snapshot of DRAM, and must hence consider snapshot consistency, this work only focuses on analysis of specific memory pages, *e.g.,* those that contain specific kernel data structures. It also cannot access CPU register state, making it vulnerable to address-translation attacks [113, 130].

5.8 Conclusion

Vendors are beginning to integrate memory and processing logic on-chip using on-package DRAM manufacturing technology. We have presented SnipSnap, an application of this technology to secure memory acquisition. SnipSnap has a hardware TCB, and allows forensic analysts to collect consistent memory snapshots from a target machine while offering performance isolation for applications executing on the target. Our experimental evaluation on a number of data intensive workloads shows the benefit of our approach.

Chapter 6

Summary

The primary motivation for the creation of virtual memory was to improve programmability [76, 77, 36]. Security, process isolation, and memory protection were added naturally to the basic abstraction. Overall, virtual memory's ability to encompasses all these features has made it vital to the success of computing.

This thesis looks at two major trends, *heterogeneity* and *"big data"*, and their interplay with the programmability and security of modern systems. Motivated by these two trends, we show how to build efficient virtual memory and complement its security with hardware techniques. We studied address translation bottlenecks and proposed solutions for CPU, GPU, and accelerator TLBs. Our first approach was to increase TLB utilization by enabling multiple page sizes to share the same (unique) TLB structure. Additionally, we enable TLB coalescing, which ultimately can compress many TLB entries into one. We achieved a larger TLB coverage for the same hardware budget, thus decreasing TLB misses. This TLB design is general and can be applied to any processing element. We evaluated our approaches with "big data" workloads for CPUs, GPUs, and virtualized CPUs.

Our second approach to improve programmability in heterogeneous systems was also motivated by the idea of higher utilization of TLBs. We observed that per-core multi-level TLBs replicate entries across per-core TLBs. Inspired by the last-level cache idea, a last-level TLB eliminates replication but suffers from higher access latency. To overcome this problem, we co-designed distributed TLBs with a SMART interconnect, which enables a low-latency TLB lookup with a shared last-level TLB design.

Our third and last approach to improve programmability applies to GPUs and throughput oriented accelerators running in a system with unified address space. TLB pressure can stall them completely due to handling address translation requests. We observed that by reordering some of the address translation requests, we could minimize the stalling periods and therefore reduce the address translation costs.

Finally, we studied mechanisms to complement VM protection. We design a memory snapshotting mechanism that provides a faithful, consistent and high-efficient copy of the physical memory, thus enabling many forensic analyses tools to perform their operations on the data and improve the security checks and guarantees we need.

All the approaches presented in this thesis help maintain the virtual memory, reducing its performance overheads and improving its security mechanisms.

Appendix A

SnipSnap TLA+ Model

MODULE *SnipSnap*

EXTENDS *Naturals*, *Integers*, *Sequences*, *FiniteSets*, *TLC*

VARIABLES *mem*, *memInt*, *ctl*, *buf*, *cache*, *cow*, *memQ*, *snapCtl*, *snapIdx*, *memt*0, *memCopied*

CONSTANT *Proc*, *Adr*, *Val*, *QLen*, *CacheSets*, *CacheWays*

ASSUME $(QLen \in Nat) \land (QLen > 0)$

3D-Cache size must be even and greater than 1

ASSUME $(CacheSets \in Nat) \land (CacheSets \geq 1)$

ASSUME $(CacheWays \in Nat) \land (CacheWays \geq 1)$ *CacheWays* has to be *power*(2)

Memory requests

$MReq \triangleq [op : \{\text{“Rd”}\}, adr : Adr]$
$\quad\quad \cup [op : \{\text{“Wr”}\}, adr : Adr, val : Val]$

$NoVal \triangleq \text{CHOOSE } v : v \notin Val$

$Init \triangleq \land mem = \text{CHOOSE } m \in [Adr \rightarrow Val] : \text{TRUE}$
$\quad\quad \land ctl \;\;= [p \in Proc \mapsto \text{“rdy”}]$
$\quad\quad \land buf \;\;= [p \in Proc \mapsto NoVal]$
$\quad\quad \land cache = [row \in (0 \,..\, CacheSets - 1) \mapsto [a \in Adr \mapsto NoVal]]$
$\quad\quad \land cow = [row \in (0 \,..\, CacheSets - 1) \mapsto [a \in Adr \mapsto NoVal]]$
$\quad\quad \land memQ = \langle\rangle$
$\quad\quad \land memInt \in \{\langle \text{CHOOSE } p \in Proc : \text{TRUE}, NoVal\rangle\}$
$\quad\quad \land snapCtl = -1$
$\quad\quad \land snapIdx = 0$
$\quad\quad \land memt0 = NoVal$
$\quad\quad \land memCopied = NoVal$

$TypeInvariant \triangleq$
$\quad \land mem \quad\;\; \in [Adr \rightarrow Val]$
$\quad \land ctl \quad\quad\; \in [Proc \rightarrow \{\text{“rdy”}, \text{“busy”}, \text{“waiting”}, \text{“done”}\}]$
$\quad \land buf \quad\quad \in [Proc \rightarrow MReq \cup Val \cup \{NoVal\}]$
$\quad \land cache \quad \in [(0 \,..\, CacheSets - 1) \rightarrow [Adr \rightarrow Val \cup \{NoVal\}]]$
$\quad \land cow \quad\;\; \in [(0 \,..\, CacheSets - 1) \rightarrow [Adr \rightarrow Val \cup \{NoVal\}]]$
$\quad \land memQ \quad \in Seq(MReq)$
$\quad \land snapCtl \;\in \{-1, 0, 1\}$
$\quad \land snapIdx \;\in Adr \cup \{NoVal\}$

$\land\ memt0 \quad \in [Adr \rightarrow Val] \cup \{NoVal\}$
$\land\ memCopied \in [Adr \rightarrow Val] \cup \{NoVal\}$

Total cache size

$CacheSize \triangleq CacheSets * CacheWays$

Cache usage

$CacheRowUsage(r) \triangleq Cardinality(\{a \in Adr : cache[r][a] \neq NoVal\})$
$RecCUsage[r \in Int] \triangleq \text{IF } r = -1 \text{ THEN } 0 \text{ ELSE } CacheRowUsage(r) + RecCUsage[r-1]$
$CacheUsage \triangleq RecCUsage[CacheSets - 1]$

CoW usage

$CoWRowUsage(r) \triangleq Cardinality(\{a \in Adr : cow[r][a] \neq NoVal\})$
$RecCoWUsage[r \in Int] \triangleq \text{IF } r = -1 \text{ THEN } 0 \text{ ELSE } CoWRowUsage(r) + RecCoWUsage[r-1]$
$CoWUsage \triangleq RecCoWUsage[CacheSets - 1]$

Safety checks

Limit the number of valid entries in the cache & *CoW* area

$CacheProp \triangleq \land\ CacheUsage + CoWUsage \leq CacheSize$
$\quad \land\ \forall r \in (0 \,..\, CacheSets - 1) : \land\ CacheRowUsage(r) \leq CacheWays$
$\quad\quad \land\ CoWRowUsage(r) \leq CacheWays$

$SnapshotConsistency \triangleq$
$\quad (snapCtl = 1 \land snapIdx = NoVal \land memt0 \neq NoVal) \implies memt0 = memCopied$

Processor $\leftarrow\ \rightarrow$ Memory system interface

$Req(p) \triangleq \land\ ctl[p] = \text{“rdy”}$

when *snapCtl* equals -1, we are not in snapshot mode
when *snapCtl* equals 0, we are in transition to snapshot mode
when *snalCtl* equals 1, we are in snapshot mode
we can't receive/process requests while transitioning from
regular mode to snapshot mode.

$\quad \land\ snapCtl \neq 0$
$\quad \land\ \exists\, req \in MReq :$
$\quad\quad \land\ memInt' = \langle p, req \rangle$ Send request to the *mem.* interface
$\quad\quad \land\ buf' = [buf \text{ EXCEPT } ![p] = req]$
$\quad\quad \land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“busy”}]$
$\quad \land\ \text{UNCHANGED } \langle mem, memQ, cache, cow, snapCtl, snapIdx, memt0, memCopied \rangle$

$Rsp(p) \triangleq \land\ ctl[p] = \text{“done”}$
$\quad \land\ memInt' = \langle p, buf[p] \rangle$ Request recv'ed
$\quad \land\ ctl' = [ctl \text{ EXCEPT } ![p] = \text{“rdy”}]$
$\quad \land\ \text{UNCHANGED } \langle mem, memQ, cache, cow, buf, snapCtl, snapIdx, memt0, memCopied \rangle$

```
Read miss − req not found in the cache, send to memory. We don't
look at the CoW area after a Read req, it may contain stale data
$RdMiss(p) \triangleq$
    LET $row \triangleq$ IF $buf[p] \neq NoVal$ THEN $(buf[p].adr \% CacheSets)$ ELSE $-1$
            CSize is the cache size based on the current mode the system is
            $CSize \triangleq$ IF $snapCtl \leq 0$ THEN $CacheSize$ ELSE $CacheSize \div 2$
    IN      $\land (ctl[p] = \text{“busy”}) \land (buf[p].op = \text{“Rd”})$
            $\land CacheUsage < CSize$  Any free CL in the cache?
            $\land CacheRowUsage(row) < CacheWays$  Any free CL in the set?
            $\land cache[row][buf[p].adr] = NoVal$
            $\land Len(memQ) < QLen$
            $\land memQ' = Append(memQ, buf[p])$
            $\land ctl' = [ctl$ EXCEPT $![p] = \text{“waiting”}]$
            $\land$ UNCHANGED $\langle memInt, mem, cache, cow, buf, snapCtl, snapIdx, memt0,$
                              $memCopied\rangle$

Write-allocate write miss - we actually perform a read operation here
Writing the address to memory is done in DoWr(p)
$WrMissRdAlloc(p) \triangleq$
    LET $row \triangleq$ IF $buf[p] \neq NoVal$ THEN $buf[p].adr \% CacheSets$ ELSE $-1$
            CSize is the cache size based on the current mode the system is
            $CSize \triangleq$ IF $snapCtl \leq 0$ THEN $CacheSize$ ELSE $CacheSize \div 2$
    IN      $\land (ctl[p] = \text{“busy”}) \land (buf[p].op = \text{“Wr”})$
            $\land CacheUsage < CSize$  Any free CL in the cache?
            $\land CacheRowUsage(row) < CacheWays$  Any free CL in the set?
            $\land cache[row][buf[p].adr] = NoVal$
            $\land Len(memQ) < QLen$
            $\land memQ' = Append(memQ, [op \mapsto \text{“Rd”}, adr \mapsto buf[p].adr])$
            $\land ctl' = [ctl$ EXCEPT $![p] = \text{“waiting”}]$
            $\land$ UNCHANGED $\langle memInt, mem, cache, cow, buf, snapCtl, snapIdx, memt0,$
                              $memCopied\rangle$

$Evict(adr) \triangleq$
    LET $row \triangleq adr \% CacheSets$
    IN   No outstanding request for address 'adr'
          $\land \forall pr \in Proc : (ctl[pr] = \text{“waiting”}) \implies (buf[pr].adr \neq adr)$
          $\land cache[row][adr] \neq NoVal$
           write back cache[row][adr] to the memory
          $\land Len(memQ) < QLen$
          $\land memQ' = Append(memQ, [op \mapsto \text{“Wr”}, adr \mapsto adr, val \mapsto cache[row][adr]])$
           clear the entry
          $\land cache' = [cache$ EXCEPT $![row][adr] = NoVal]$
          $\land$ UNCHANGED $\langle memInt, mem, buf, ctl, cow, snapCtl, snapIdx, memt0,$
                            $memCopied\rangle$

DoRd requests always get data from the cache
RdMiss will bring data from the memory to the cache
$DoRd(p) \triangleq$
```

```
        LET $row \triangleq buf[p].adr \% CacheSets$
            $adr \triangleq buf[p].adr$
        IN  $\land ctl[p] \in \{$"busy", "waiting"$\}$
            $\land buf[p].op =$ "Rd"
            $\land cache[row][adr] \neq NoVal$
            $\land buf' = [buf$ EXCEPT $![p] = cache[row][adr]]$
            $\land ctl' = [ctl$ EXCEPT $![p] =$ "done"$]$
            $\land$ UNCHANGED $\langle memInt, mem, memQ, cache, cow, snapCtl, snapIdx,$
                             $memt0, memCopied\rangle$

$DoWr(p) \triangleq$
    LET $r \triangleq buf[p]$
        $row \triangleq buf[p].adr \% CacheSets$
        $CSize$ is the cache size based on the current mode the system is
        $CSize \triangleq$ IF $snapCtl \leq 0$ THEN $CacheSize$ ELSE $CacheSize \div 2$
    IN  $\land ctl[p] \in \{$"busy", "waiting"$\}$
        $\land CoWUsage < CSize$    We can't write if $CoW$ is full
        $\land CoWRowUsage(row) < CacheWays$   We can't write if $CoW$'s set is full
        $\land (r.op =$ "Wr"$)$
        $\land cache[row][r.adr] \neq NoVal$
        $\land cow' =$ IF $\land snapCtl = 1$
                         $\land snapIdx \neq NoVal$
                         $\land r.adr \geq snapIdx$
                         $\land cow[row][r.adr] = NoVal$
                      THEN $[cow$ EXCEPT $![row][r.adr] = cache[row][r.adr]]$
                      ELSE $cow$
        $\land cache' =$ IF $cache[row][r.adr] \neq NoVal$
                        THEN $[cache$ EXCEPT $![row][r.adr] = r.val]$
                        ELSE $cache$
        $\land buf' = [buf$ EXCEPT $![p] = NoVal]$
        $\land ctl' = [ctl$ EXCEPT $![p] =$ "done"$]$
        $\land$ UNCHANGED $\langle memInt, mem, memQ, snapCtl, snapIdx, memt0,$
                         $memCopied\rangle$

Memory/cache operation
$vmem$ is a helper function used in $MemQRd$
$vmem \triangleq$
  LET $f[i \in 0 \,..\, Len(memQ)] \triangleq$
        IF $i = 0$ THEN $mem$
                ELSE IF $memQ[i].op =$ "Rd"
                       THEN $f[i-1]$
                       ELSE $[f[i-1]$ EXCEPT $![memQ[i].adr] =$
                                                $memQ[i].val]$
  IN  $f[Len(memQ)]$

$MemQWr \triangleq$ LET $r \triangleq Head(memQ)$
              IN  $\land (memQ \neq \langle\rangle) \land (r.op =$ "Wr"$)$
                  $\land mem' = [mem$ EXCEPT $![r.adr] = r.val]$
                  $\land memQ' = Tail(memQ)$
```

```
                 $\land$ UNCHANGED $\langle memInt, buf, ctl, cache, cow, snapCtl, snapIdx,$
                                   $memt0, memCopied\rangle$

$MemQRd \triangleq$
  LET $r \triangleq Head(memQ)$
      $row \triangleq r.adr \% CacheSets$
  IN  $\land (memQ \neq \langle\rangle) \land (r.op = \text{"Rd"})$
      $\land memQ' = Tail(memQ)$
      $\land cache' = [cache$ EXCEPT $![row][r.adr] = vmem[r.adr]]$
      $\land$ UNCHANGED $\langle memInt, mem, buf, ctl, cow, snapCtl, snapIdx,$
                        $memt0, memCopied\rangle$

Request the snapshot
$StartSnapReq \triangleq \land snapCtl = -1$
                 $\land snapCtl' = 0$
                 $\land snapIdx' = 0$
                 $\land$ UNCHANGED $\langle memInt, mem, memQ, buf, ctl, cache, cow,$
                                   $memt0, memCopied\rangle$

Wait until all outstanding *mem* requests are finished
so we can flush the cache, and effectively start the snapshot
$StartSnapAck \triangleq$
    $\land snapCtl = 0$
    No more outstanding memory requests
    $\land \forall p \in Proc : ctl[p] = \text{"rdy"}$
    All the cache entries have been written-back
    $\land \forall row \in (0 \,..\, CacheSets - 1) : \forall a \in Adr : cache[row][a] = NoVal$
    Flush the cache and cow
    $\land cache' = [row \in (0 \,..\, CacheSets - 1) \mapsto [a \in Adr \mapsto NoVal]]$
    $\land cow' = [row \in (0 \,..\, CacheSets - 1) \mapsto [a \in Adr \mapsto NoVal]]$
    Flush the memory queue
    $\land Len(memQ) = 0$
    Switch to snapshot mode
    $\land snapCtl' = snapCtl + 1$
    make a copy of the memory to verify if our snapshot method
    $\land memt0' = mem$
    $\land memCopied' =$ CHOOSE $m \in [Adr \rightarrow Val]$ : TRUE
    $\land$ UNCHANGED $\langle memInt, mem, memQ, buf, ctl, snapIdx\rangle$

Taking the snapshot $-$ *increaments snapIdx* as we perform the snapshot
We also evict entries in the *CoW* area that have already been copied
$TakingSnap \triangleq$
    LET $row \triangleq snapIdx \% CacheSets$
    IN  $\land snapCtl > 0$
        $\land snapIdx \neq NoVal$
        $\land snapIdx < Cardinality(Adr)$   *snapIdx* is zero-based
        $\land memCopied' =$ CASE $cow[row][snapIdx] \neq NoVal \rightarrow$
                         $[memCopied$ EXCEPT $![snapIdx] = cow[row][snapIdx]]\Box$
                           $\land cache[row][snapIdx] \neq NoVal$
```

```
                          $\land\ cow[row][snapIdx] = NoVal \rightarrow$
                         $[memCopied\ \text{EXCEPT}\ ![snapIdx] = cache[row][snapIdx]]\Box$
                     $\text{OTHER}\ \rightarrow$
                           $[memCopied\ \text{EXCEPT}\ ![snapIdx] = mem[snapIdx]]$
        $\land\ snapIdx' = \text{IF}\ snapIdx < Cardinality(Adr) - 1$
                           $\text{THEN}\ snapIdx + 1$
                           $\text{ELSE}\ NoVal$
        $\land\ cow' = [cow\ \text{EXCEPT}\ ![row][snapIdx] = NoVal]$
        $\land\ \text{UNCHANGED}\ \langle memInt, mem, memQ, buf, ctl, cache, snapCtl, memt0\rangle$

$Next \triangleq\ \lor\ \exists\, p \in Proc : \lor\ Req(p) \lor Rsp(p)$
                            $\lor\ RdMiss(p) \lor WrMissRdAlloc(p)$
                            $\lor\ DoRd(p) \lor DoWr(p)$
           $\lor\ \exists\, a \in Adr\ : Evict(a)$
           $\lor\ MemQWr \lor MemQRd$
           $\lor\ StartSnapReq \lor StartSnapAck \lor TakingSnap$

$Spec \triangleq$
  $Init \land \Box[Next]_{\langle memInt, mem, memQ, buf, ctl, cache, cow, snapIdx, snapCtl, memt0, memCopied\rangle}$
```

THEOREM $Spec \implies \Box(TypeInvariant \land CacheProp \land SnapshotConsistency)$

References

[1] CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[2] Docker – Build, Ship and Run Any App, Anywhere. https://www.docker.com/.

[3] eDRAM. Wikipedia entry: https://en.wikipedia.wiki/EDRAM.

[4] The gem5 simulator. http://gem5.org/.

[5] Graph500. http://www.graph500.org.

[6] IOMMU tutorial at ASPLOS 2016. http://pages.cs.wisc.edu/~basu/isca_iommu_tutorial/IOMMU_TUTORIAL_ASPLOS_2016.pdf.

[7] IOMMU v2 specification. https://developer.amd.com/wordpress/media/2012/10/488821.pdf.

[8] Memcached. https://memcached.org.

[9] Rekall Forensics – We can remember it for you wholesale! http://www.rekall-forensic.com/.

[10] Shortest job first scheduling. http://www.geeksforgeeks.org/operating-system-shortest-job-first-scheduling-predicted-burst-time/.

[11] SPEC. https://www.spec.org/cpu2006/.

[12] Tensorflow. https://www.tensorflow.org.

[13] Unified memory in CUDA 6. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[14] Volatility – An advanced memory forensics framework. https://github.com/volatilityfoundation/volatility.

[15] ARM security technology – Building a secure system using TrustZone technology, 2009. ARM Technical Whitepaper. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[16] Wind River Simics, 2015. https://www.windriver.com/products/simics/.

[17] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch. Unlocking bandwidth for GPUs in CC-NUMA systems. In *IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA '15, pages 354–365, 2015.

[18] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. 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, 2015.

[19] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[20] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, 2015.

[21] W. Arbaugh. Komoku. `https://www.cs.umd.edu/~waa/UMD/Home.html`.

[22] A. Arcangeli. Transparent Hugepage Support. *KVM Forum*, 2010.

[23] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. 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, 2012.

[24] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu. Mask: Redesigning the GPU memory hierarchy to support multi-application concurrency. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 503–518, 2018.

[25] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS '14, pages 90–102, 2014.

[26] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Secur. Comput.*, 8(5):670–684, Sept. 2011.

[27] T. W. Barr, A. L. Cox, and S. 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, 2010.

[28] T. W. Barr, A. L. Cox, and S. Rixner. SpecTLB: A mechanism for speculative address translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 307–318, 2011.

[29] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. 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, 2013.

[30] A. Basu, M. D. Hill, and M. 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, 2012.

[31] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee. Scalable distributed shared last-level TLBs using low-latency interconnects. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (to appear)*, MICRO-51, 2018.

[32] R. Bhargava, B. Serebrin, F. Spadini, and S. 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, 2008.

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

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

[35] A. 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, 2017.

[36] A. Bhattacharjee and D. Lustig. Architectural and operating system support for virtual memory. *Synthesis Lectures on Computer Architecture*, 12(5):1–175, 2017.

[37] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level TLBs for chip multiprocessors. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 62–63, 2011.

[38] A. Bhattacharjee and M. Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 29–40, 2009.

[39] A. Bhattacharjee and M. Martonosi. Inter-core cooperative TLB for chip multiprocessors. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 359–370, 2010.

[40] C. Bienia, S. Kumar, J. P. Singh, and K. 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, 2008.

[41] M. Bilzor. 3D execution monitor (3D-EM): Using 3D circuits to detect hardware malicious inclusions in general purpose processors. In *6th International Conference on Information Warfare and Security*, 2011.

[42] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Security checkers: Detecting processor malicious inclusions at runtime. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 34–39, 2011.

[43] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 49–54, 2012.

[44] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-39, pages 469–479, 2006.

[45] E. Blem, M. Sinclair, and K. Sankaralingam. Challenge benchmarks that must be conquered to sustain the GPU revolution. In *Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture*, 2011.

[46] F. Bodin and A. Seznec. Skewed associativity enhances performance predictability. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 265–274, 1995.

[47] A. Bohra, I. Neamtiu, and F. Sultan. Remote repair of operating system state using backdoors. In *Proceedings of the First International Conference on Autonomic Computing*, ICAC '04, pages 256–263, 2004.

[48] J. Buell, D. Hecht, J. Heo, K. Saladi, and R. Taheri. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. *VMWare Technical Journal*, 2(1):19–28, Jun 2013.

[49] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '12, pages 141–151, 2012.

[50] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 555–565, 2009.

[51] A. Case and G. G. Richard. Memory forensics: The path forward. *Digital Investigation*, 20:23 – 33, 2017. Special Issue on Volatile Memory Analysis.

[52] M. Chan, H. Litz, and D. R. Cheriton. Rethinking network stack design with memory snapshots. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS '13, pages 27–27, 2013.

[53] S. Chan. A Brief Intro to the Heterogeneous Compute Compiler, 2016. `https://gpuopen.com/a-brief-intro-to-boltzmann-hcc/`.

[54] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi. Staged reads: Mitigating the impact of DRAM writes on DRAM reads.

In *Proceedings of the IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, 2012.

[55] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian. Managing DRAM latency divergence in irregular GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 128–139, 2014.

[56] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis. Improving sha-2 hardware implementations. In *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, CHES '06, pages 298–310, 2006.

[57] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph applications. In *IEEE International Symposium on Workload Characterization*, IISWC '13, pages 185–195, 2013.

[58] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54, 2009.

[59] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '10, pages 1–11, 2010.

[60] C.-H. O. Chen, S. Park, T. Krishna, S. Subramanian, A. P. Chandrakasan, and L.-S. Peh. SMART: A single-cycle reconfigurable NoC for SoC applications. In *2013 Design, Automation Test in Europe Conference Exhibition*, DATE '13, pages 338–343, 2013.

[61] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze. 14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE International Solid-State Circuits Conference*, ISSCC '16, pages 262–263, 2016.

[62] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi. Hicamp: Architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 287–300, 2012.

[63] C. Chou, A. Jaleel, and M. 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, 2014.

[64] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, 2012.

[65] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 308–319, 2013.

[66] G. Cox and A. 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, 2017.

[67] G. Cox, Z. Yan, A. Bhattacharjee, and V. Ganapathy. Secure, consistent, and high-performance memory snapshotting. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 236–247, 2018.

[68] L. Cui, T. Wo, B. Li, J. Li, B. Shi, and J. Huai. PARS: A page-aware replication system for efficiently storing virtual machine snapshots. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 215–228, 2015.

[69] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 820–831, 2016.

[70] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of the 21st USENIX Conference on Security Symposium*, SEC '12, pages 42–42, 2012.

[71] CVE-2007-4993. Xen guest root escapes to dom0 via pygrub.

[72] CVE-2007-5497. Integer overflows in libext2fs in e2fsprogs.

[73] CVE-2008-0923. Directory traversal vulnerability in the shared folders feature for VMWare.

[74] CVE-2008-1943. Buffer overflow in the backend of XenSource Xen paravirtualized frame buffer.

[75] CVE-2008-2100. VMWare buffer overflows in VIX API let local users execute arbitrary code in host OS.

[76] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, Sept. 1970.

[77] P. J. Denning. Before memory was virtual. In *In the Beginning: Personal Recollections of Software Pioneers*, pages 250–271. Wiley-IEEE Computer Society Press, 1997.

[78] A. M. Devices. Magic packet technology, November 1995. AMD Publication number 20213, `https://support.amd.com/TechDocs/20213.pdf`.

[79] B. Egger, E. Gustafsson, C. Jo, and J. Son. Efficiently restoring virtual machines. *International Journal of Parallel Programming*, 43(3):421–439, 2015.

[80] D. Fan, Z. Tang, H. Huang, and G. R. Gao. An energy efficient TLB design methodology. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 351–356, 2005.

[81] Q. Feng, A. Prakash, H. Yin, and Z. Lin. Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 196–205, 2014.

[82] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. 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, 2012.

[83] H. Fujita, N. Dun, Z. A. Rubenstein, and A. A. Chien. Log-structured global array for efficient multi-version snapshots. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 281–291, 2015.

[84] J. Gandhi, A. Basu, M. D. Hill, and M. 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, 2014.

[85] J. Gandhi, M. D. Hill, and M. 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, 2016.

[86] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Range translations for fast virtual memory. *IEEE Micro*, 36(3):118–126, 2016.

[87] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Symposium on Network and Distributed System Security*, NDSS '03, pages 191–206, 2003.

[88] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema. Large pages may be harmful on NUMA systems. In *2014 USENIX Annual Technical Conference*, USENIX ATC 14, pages 231–242, Philadelphia, PA, 2014.

[89] X. Ge, H. Vijayakumar, and T. Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. In *Proceedings of the Third Workshop on Mobile Security Technologies*, MoST '14, 2014.

[90] Google. Using DDMS for debugging. `http://developer.android.com/tools/debugging/ddms.html`.

[91] M. Graziano, A. Lanzi, and D. Balzarotti. Hypervisor memory forensics. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID 2013, pages 21–40, 2013.

[92] K. Group. OpenCL, 2014. `https://www.khronos.org/opencl/`.

[93] A. Gundu, A. S. Ardestani, M. Shevgoor, and R. Balasubramonian. A case for near data security. In *2nd Workshop on Near Data Processing*, WoNDP-2, 2014.

[94] F. Guo, S. Kim, Y. Baskakov, and I. 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, 2015.

[95] Y. Hao, Z. Fang, G. Reinman, and J. Cong. Supporting address translation for accelerator-centric architectures. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA '17, pages 37–48, 2017.

[96] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 184–195, 2009.

[97] M. B. Healy, K. Athikulwongse, R. Goel, M. M. Hossain, D. H. Kim, Y.-J. Lee, D. L. Lewis, T.-W. Lin, C. Liu, M. Jung, B. Ouellette, M. Pathak, H. Sane, G. Shen, D. H. Woo, X. Zhao, G. H. Loh, H. H. S. Lee, and S. K. Lim. Design and analysis of 3D-MAPS: A many-core 3D processor with stacked memory. In *IEEE Custom Integrated Circuits Conference 2010*, pages 1–4, 2010.

[98] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 6th edition, 2017.

[99] A. P. Heriyanto. Procedures and tools for acquisition and analysis of volatile memory on Android smartphones. In *Proceedings of the 11th Australian Digital Forensics Conference*, pages 84–95, 2013.

[100] R. Ho, K. Mai, and M. Horowitz. Managing wire scaling: a circuit perspective. In *Proceedings of the IEEE International Interconnect Technology Conference (Cat. No.03TH8695)*, pages 177–179, 2003.

[101] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 279–290, 2011.

[102] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 204–216, 2016.

[103] Y. Huang, R. Yang, L. Cui, T. Wo, C. Hu, and B. Li. VMCSnap: Taking snapshots of virtual machine cluster with memory deduplication. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 314–319, 2014.

[104] T. Huffmire, T. Levin, M. Bilzor, C. E. Irvine, J. Valamehr, M. Tiwari, T. Sherwood, and R. Kastner. Hardware trust implications of 3-D integration. In *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS '10, pages 1:1–1:10, 2010.

[105] T. Huffmire, T. Levin, C. Irvine, R. Kastner, and T. Sherwood. 3-D extensions for trustworthy systems. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2011.

[106] T. Huffmire, J. Valamehr, T. Sherwood, R. Kastner, T. Levin, T. D. Nguyen, and C. Irvine. Trustworthy system security through 3-D integrated hardware. In *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 91–92, 2008.

[107] F. Imeson, A. Emtenan, S. Garg, and M. V. Tripunitara. Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC '13, pages 495–510, 2013.

[108] InfiniBand. The InfiniBand trade association—the InfiniBand$^{\text{TM}}$ architecture specification. `http://www.infinibandta.org`.

[109] Intel. Haswell microarchitecture. *www.7-cpu.com/cpu/Haswell.html.*

[110] Intel. Skylake microarchitecture. *www.7-cpu.com/cpu/Skylake.html.*

[111] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. 2018.

[112] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, pages 151–162, 2010.

[113] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang. Atra: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 167–178, 2014.

[114] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 850–855, 2012.

[115] N. E. Jerger, T. Krishna, and L.-S. Peh. On-chip networks, second edition. *Synthesis Lectures on Computer Architecture*, 12(3):1–210, 2017.

[116] D. Jevdjic, G. H. Loh, C. Kaynak, and B. 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, 2014.

[117] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of*

the 40th Annual International Symposium on Computer Architecture, ISCA '13, pages 404–415, 2013.

[118] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent GPGPU applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 1:1–1:8, 2014.

[119] Joint Test Action Group (JTAG). 1149.1-2013 - IEEE Standard for test access port and boundary-scan architecture, 2013. `http://standards.ieee.org/findstds/standard/1149.1-2013.html`.

[120] T. Juan, T. Lang, and J. J. Navarro. Reducing TLB power requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 196–201, 1997.

[121] 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, 2002.

[122] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for TLB prefetching: An application-driven study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 195–206, 2002.

[123] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. 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, 2015.

[124] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Energy-efficient address translation. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, HPCA '16, pages 631–643, 2016.

[125] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 157–166, 2013.

[126] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 211–222, 2002.

[127] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: A cost-efficient topology for high-radix networks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 126–137, 2007.

[128] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, HPCA '10, pages 1–12, 2010.

[129] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, pages 65–76, 2010.

[130] Y. Kinebuchi, S. Butt, V. Ganapathy, L. Iftode, and T. Nakajima. Monitoring integrity using limited local memory. *Trans. Info. For. Sec.*, 8(7):1230–1242, July 2013.

[131] Kortchinsky, Kostya. Cloudburst: Hacking 3D (and breaking out of VMWare). In *Black Hat USA*, 2009.

[132] T. Krishna, C.-H. O. Chen, W. C. Kwon, and L.-S. Peh. Breaking the on-chip latency barrier using smart. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pages 378–389, 2013.

[133] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. 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, 2016.

[134] G. Kyriazis. Heterogeneous System Architecture: A Technical Review. *Whitepaper*, 2012.

[135] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Pearson Education, 2002.

[136] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC '13, pages 511–526, 2013.

[137] J. Lee, M. Samadi, and S. Mahlke. VAST: The illusion of a large memory space for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 443–454, 2014.

[138] D. Li and T. M. Aamodt. Inter-core locality aware memory scheduling. *IEEE Comput. Archit. Lett.*, 15(1):25–28, 2016.

[139] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder. Priority-based cache allocation in throughput processors. In *IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA '15, pages 89–100, 2015.

[140] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The mcpat framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Trans. Archit. Code Optim.*, 10(1):5:1–5:29, Apr. 2013.

[141] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi. CPU transparent protection of os kernel and hypervisor integrity with programmable dram. In *Proceedings of*

the 40th Annual International Symposium on Computer Architecture, ISCA '13, pages 392–403, 2013.

[142] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464. IEEE Computer Society, 2008.

[143] G. H. Loh. Extending the effectiveness of 3d-stacked DRAM caches with an adaptive multi-queue policy. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, pages 201–212, 2009.

[144] G. H. Loh and M. 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, 2011.

[145] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[146] D. Lustig, G. Sethi, M. Martonosi, and A. 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, 2016.

[147] Y. Marathe, N. Gulur, J. H. Ryoo, S. Song, and L. K. John. CSALT: Context switch aware large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50, pages 449–462, 2017.

[148] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty. XvMotion: Unified virtual machine migration over long distance. In *2014 USENIX Annual Technical Conference*, USENIX ATC 14, pages 97–108, 2014.

[149] D. Megas, K. Pizolato, T. Levin, and T. Huffmire. A 3D data transformation processor. In *Proceedings of the Workshop on Embedded Systems Security*, WESS '12, 2012.

[150] Mellanox Technologies. Introduction to InfiniBand, September 2014. `http://www.mellanox.com/blog/2014/09/introduction-to-infiniband`.

[151] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 235–246, 2010.

[152] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS '12, pages 28–37, 2012.

[153] A. Moser and M. I. Cohen. Hunting in the enterprise: Forensic triage and incident response. *Digital Investigation*, 10(2):89 – 98, 2013. Triage in Digital Forensics.

[154] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 3–14, 2007.

[155] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 146–160, 2007.

[156] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 63–74, 2008.

[157] S. Mysore, B. Agrawal, N. Srivastava, S.-C. Lin, K. Banerjee, and T. Sherwood. Introspective 3D chips. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 264–273, 2006.

[158] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 89–104, 2002.

[159] NVIDIA. GPU-accelerated applications, 2016. `http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf`.

[160] M. Oskin and G. H. Loh. A software-managed approach to die-stacked DRAM. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 188–200, 2015.

[161] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-based superpage-friendly TLB designs. In *IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA '15, pages 210–222, 2015.

[162] M. Parasar, A. Bhattacharjee, and T. Krishna. SEESAW: Using superpages to improve VIPT caches. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 193–206, 2018.

[163] C. H. Park, T. Heo, J. Jeong, and J. 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, 2017.

[164] E. Park, B. Egger, and J. Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 75–86, 2011.

[165] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium*, SEC '04, pages 13–13, 2004.

[166] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, SEC '06, 2006.

[167] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 103–115, 2007.

[168] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, pages 558–567, 2014.

[169] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced large-reach TLBs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 258–269, 2012.

[170] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-48, pages 1–12, 2015.

[171] B. Pham, J. Vesely, G. H. Loh, and A. Bhattacharjee. Using TLB Speculation to Overcome Page Splintering in Virtual Machines. *Rutgers Technical Report DCS-TR-713*, 2015.

[172] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 743–758, 2014.

[173] B. Pichai, L. Hsu, and A. Bhattacharjee. Address translation for throughput-oriented accelerators. *IEEE Micro*, 35(3):102–113, 2015.

[174] L.-N. Pouchet and T. Yuki. Polybench, 2010. `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.

[175] J. Power, M. D. Hill, and D. A. Wood. Supporting x86-64 address translation for 100s of GPU lanes. In *IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, pages 568–578, 2014.

[176] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 235–246, 2012.

[177] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, 2000.

[178] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 371–382, 2009.

[179] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, 2012.

[180] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 99–110, 2013.

[181] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin. Specifying and dynamically verifying address translation-aware memory consistency. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 323–334, 2010.

[182] J. Rutkowska. Beyond the CPU: Defeating hardware based RAM acquisition, part I: AMD case. In *Black Hat USA*, 2007.

[183] J. Rutkowska and R. Wojtczuk. Preventing and detecting Xen hypervisor subversions. In *Black Hat USA*, 2008.

[184] J. H. Ryoo, N. Gulur, S. Song, and L. K. John. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 469–480, 2017.

[185] R. Sampson and T. F. Wenisch. ZCache Skew-ered. In *9th Annual Workshop on Duplicating, Deconstructing, and Debunking*, WDDD '11, 2011.

[186] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, pages 187–198. IEEE Computer Society, 2010.

[187] D. Sanchez and C. Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 57–68, 2011.

[188] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 427–428, 2012.

[189] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-based TLB preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 117–127, 2000.

[190] K. Saur, M. Hicks, and J. S. Foster. C-strider: Type-aware heap traversal for c. *Softw. Pract. Exper.*, 46(6):767–788, June 2016.

[191] B. Schatz and M. Cohen. Advances in volatile memory forensics. *Digital Investigation*, 20:1, 2017. Special Issue on Volatile Memory Analysis.

[192] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 169–178, 1993.

[193] A. Seznec. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Trans. Comput.*, 53(7):924–927, July 2004.

[194] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 89–101, 2015.

[195] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu. Scheduling page table walks for irregular GPU applications. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 180–192, 2018.

[196] A. Sodani. Race to Exascale: Opportunities and Challenges. *MICRO-44 Keynote*, 2011.

[197] S. Srikantaiah and M. Kandemir. Synergistic TLBs for high performance address translation in chip multiprocessors. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, pages 313–324, 2010.

[198] A. Stevenson. Boot into recovery mode for rooted and un-rooted Android devices. `http://androidflagship.com/605-enter-recovery-mode-rooted-un-rooted-android`.

[199] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia. Trustdump: Reliable memory acquisition on smartphones. In *Computer Security - ESORICS 2014*, pages 202–218, 2014.

[200] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from Android devices. *Digital Investigation*, 8(3):175–184, 2012.

[201] M. Talluri and M. 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, 1994.

[202] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 415–424, 1992.

[203] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 22:1–22:11, 2009.

[204] Tezzaron Semiconductors. 3D-ICs and integrated circuit security, 2008. `http://www.tezzaron.com/media/3D-ICs_and_Integrated_Circuit_Security.pdf`.

[205] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. XSBench - the development and verification of a performance abstraction for monte carlo reactor analysis. In *The Role of Reactor Physics toward a Sustainable Future*, PHYSOR 2014.

[206] H. Usui, L. Subramanian, K. K.-W. Chang, and O. Mutlu. DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans. Archit. Code Optim.*, 12(4):65:1–65:28, 2016.

[207] J. Valamehr, T. Huffmire, C. Irvine, R. Kastner, Ç. K. Koç, T. Levin, and T. Sherwood. A qualitative security analysis of a new class of 3-D integrated crypto co-processors. In *Cryptography and Security: From Theory to Applications: Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*, pages 364–382. Springer Berlin Heidelberg, 2012.

[208] J. Valamehr, T. Sherwood, R. Kastner, D. Marangoni-Simonsen, T. Huffmire, C. Irvine, and T. Levin. A 3-d split manufacturing approach to trustworthy system development. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 32(4):611–615, 2013.

[209] J. Valamehr, M. Tiwari, T. Sherwood, R. Kastner, T. Huffmire, C. Irvine, and T. Levin. Hardware assistance for trustworthy systems through 3-d integration. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 199–210, 2010.

[210] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. 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 '16, pages 161–171, 2016.

[211] B. Wang, W. Yu, X.-H. Sun, and X. Wang. Dacache: Memory divergence-aware GPU cache management. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 89–98, 2015.

[212] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In *IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, pages 225–236, 2014.

[213] Wang, Bin. Mitigating GPU memory divergence for data-intensive applications, 2015. `https://etd.auburn.edu/bitstream/handle/10415/4688/dissertation.pdf?sequence=2&isAllowed=y`.

[214] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee. Hardware translation coherence for virtualized systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 430–443, 2017.

[215] H. Yoon, J. Lowe-Power, and G. S. Sohi. Filtering translation bandwidth with virtual caching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 113–127. ACM, 2018.

[216] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for GPUs. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA '16, pages 345–357, 2016.

[217] R. Zhou and T. Li. Leveraging phase change memory to achieve efficient virtual machine execution. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 179–190, 2013.

URLs in references were last accessed September 26, 2018