ARCHITECTURAL SUPPORT FOR EFFICIENT VIRTUAL MEMORY ON BIG-MEMORY SYSTEMS.

BY BINH QUANG PHAM

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Abhishek Bhattacharjee

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2016

ABSTRACT OF THE DISSERTATION

Architectural Support for Efficient Virtual Memory on Big-Memory Systems.

by Binh Quang Pham

Dissertation Director: Abhishek Bhattacharjee

Virtual memory is a powerful and ubiquitous abstraction for managing memory. However, virtual memory suffers a performance penalty for these benefits, namely when translating program virtual addresses to system physical addresses. This overhead had been limited to 5-15% of system runtime by using a set of sophisticated hardware solutions, but has increased to 20-50% for many scenarios, including running workloads with large memory footprints and poor access locality or using deeper software stacks.

My thesis aims to solve this problem so that the memory systems can continue to scale without being hamstrung by the virtual memory system. We observe that while operating systems (OS) and hypervisors have a rich set of components in allocating memory, the hardware address translation unit only maintains a rigid and limited view of this ecosystem. Therefore, we seek for patterns inherently present in the memory allocation mechanisms to guide us in designing a more intelligent address translation unit.

First, we realize that OS memory allocators and program faulting sequence tend to produce contiguous or nearby mappings between virtual and physical pages. We propose Coalesced TLB and Clustered TLB designs to exploit these patterns accordingly. Once detected, the related mappings are stored in a single TLB entry to increase the TLB

reach. Our designs help reduce TLB misses substantially and improve performance as a result.

Second, we see that there are often tradeoffs between reducing address translation overheard and improving resource consolidation in virtualized environments. For example, large pages are often used to mitigate the high cost of two-dimensional page walks, but hypervisors usually break large pages into small pages for easier sharing guests memory. When that happens, the majority of those small pages still remain aligned. Based on this observation, we propose a speculative TLB technique to regain almost all performance loss caused by breaking large pages while running highly consolidated virtualized systems.

Acknowledgements

Seven years ago, I was very happy when I received an admission letter to pursue a PhD degree in the US. Fresh out of college, I had little idea of what my research area would be and what doing a PhD entailed. Looking back, I find that doing a PhD is one of the most challenging yet rewarding experience of my life. As I am coming to the end of my PhD journey, I would like to take a moment and acknowledge the many people that I am thankful for in my life.

First and foremost, I wish to thank my advisor, Dr. Abhishek Bhattacharjee for his guidance in developing this thesis. I am grateful to him for teaching me how to do research, from formulating ideas, conducting experiments, to presenting the final results. We have had many discussions in the last few years working together – I always came out with a clear idea of what to do next and felt strongly motivated to go to the end of the road after each and every discussion.

I owe a very special thank you to my parents for their unconditional love and encouragement throughout my life. To my sisters, Thuy and Ha, my brother-in-laws Kien, Hieu, my nephews and niece for their love and support. To Hue for always being there for me and sharing the ups and downs of a PhD student's life. Whenever I feel tired, thinking of them gives me strength to continue and try.

I also wish to thank:

- Dr. Gabriel H. Loh for being a great mentor and collaborator during and after my internship at AMD. I am fortunate enough to work with him on a major part of my thesis, and I learn a great deal from him, whether it is understanding technical concepts, generating ideas, or getting work done in the most efficient way

- Dr. Thu Nguyen, Dr. Ricardo Bianchini, and Dr. Martha Kim for serving as my committee members and their valuable feedback while I am working on this

thesis

- Dr. Yasuko Eckert and Dr. Trey Cain for being amazing mentors during my internships at AMD and Qualcomm. These internship experiences help me better understand how research ideas can and should be applied to the outside world
- Members and former members of the RUARCH Lab for helping me complete the projects that lead to this thesis, for attending endless practice talks and proofreading my paper submissions as well as this thesis
- Department of Computer Science at Rutgers University for being my second home in the US.

Dedication

To my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation shows that significant overhead is spent handling address translation, but the majority of this overhead can be eliminated by exploiting prevalent patterns present in memory allocation mechanisms.

1.1 Motivation

As processor vendors embrace the era of big data, fields like scientific computing, data mining, social networks, and business management depend on processing massive, multi-dimensional data sets. In this context, it is critical to re-evaluate virtual memory, ubiquitous across computer systems today since it is a powerful abstraction for allocating and managing memory with a flexible and clean programming model. Specifically, virtual memory allows programmers to fully concentrate on finding solutions to their problems as they do not have to worry about the underlying physical memory layout as well as data movement between main memory and backing storage. Virtual memory also provides isolation and protection between processes as each process operates in its own address space. Furthermore, virtual memory promotes modularity in development of complex programs, where individual modules in these programs can be compiled separately and only linked together at runtime. Virtual memory achieves these benefits by separating the virtual address space, which consists of identifiers used by programs to reference information, from the physical address space, which contains memory locations where information is actually stored.

Unfortunately, virtual memory suffers from a performance tax caused by this abstraction layer, namely when translating virtual addresses to physical addresses. Therefore, it is crucial to keep this overhead as small as possible so that all of the programmability benefits provided by virtual memory remain accessible to programmers. In fact, according to Peter J. Denning, a pioneer in the field of virtual memory, address translation overhead should stay within 3% of hardware execution time in order for virtual memory systems to be widely adopted by programmers [29].

To keep this overhead as small as possible, hardware vendors have come up with a set of sophisticated solutions, including Memory Management Unit (MMU) with multilevel TLBs, MMU caches, and hardware page table walkers tightly integrated with the core pipelines and memory systems. Despite these efforts, translation overhead remains high at 5% - 14% of system runtime [19,44]. The complex and multi-step translation process attributes to this high overhead. At a high level, mappings between virtual addresses and physical addresses are stored in page tables and managed by the operating system (OS). Page tables are usually organized in a radix-tree format [1,2,5] with many levels. Therefore, translating a virtual to physical address would involve multiple memory accesses, which take several hundreds of CPU cycles to complete [44]. As fast address translation is critical to virtual memory performance, modern processors usually rely on Translation Lookaside Buffers (TLBs), which are on-chip, content-addressable caches, to keep the most recently used virtual-to-physical mappings. Hits in TLBs often return the physical address in a few clock cycles, whereas misses in TLBs default to the long latency multi-level page table look up. Therefore, avoiding TLB misses is key to virtual memory performance. Unfortunately, there are several noticeable trends in the memory systems that tend to increase TLB misses and worsen the translation overhead as discussed below.

First, application trends are aggressively pushing toward larger and larger memory requirements; to date about 2.5 quintillion bytes of data are created daily [41]. From large traditional databases to Internet-driven content, from graph analytics to bioinformatics, the need to process and analyze huge amount of information is accelerating [16,32]. However, performance challenges associated with scaling up virtual memory

to support these growing application areas may significantly slow down progress.

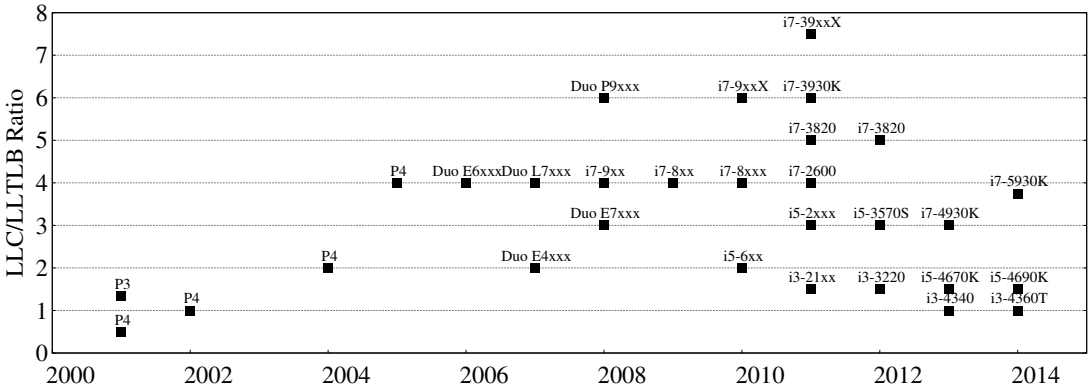


Figure 1.1: Ratio between last level cache capacity and last level TLB reach.

Second, the move toward expanding cache and physical memory size further increases pressure on virtual memory systems and in particular, TLBs. Figure 1.1 plots the ratio between the last level cache and TLB capacity in recent Intel processors. The gap between cache and TLB capacity increased steadily from 2000 to 2011, with the maximum ratio of 7x seen in core i7, 2011. Even though processor vendors have tried to mitigate this gap, it remains high, currently at 4x in core i7, 2014. Besides, the adoption of 3D stacked DRAM [40, 46, 62] and its use as a DRAM cache [47, 55, 56, 71, 81] only worsen this ratio. Similarly, we have seen enormous efforts in architecting future memory technologies [27, 43, 53, 54, 89] which make terabytes of memory possible. This in turn puts tremendous pressure on the reach of any caching structures, including TLBs, in modern processors.

Third, systems with many software layers, e.g. virtualization, are becoming more popular [3]. The main benefits of using such systems are good process isolation and resource consolidation. However, address translation overhead in these scenarios is likely to increase significantly compared to native execution [7,19,26,34]. In fact, according to a recent VMware study, TLB miss handling cost due to the hardware-assisted memory management unit is the largest contributor to the performance difference between native and virtual execution [26].

Fourth, the move toward supporting fully-shared virtual memory between CPUs

and GPUs (for example, as specified in the Heterogeneous Systems Architecture specification [52, 72]) further increases the pressure on virtual memory systems and TLBs. The need to provide address translation services for potentially thousands of threads will require architects to squeeze as much benefit out of every last bit of SRAM spent on implementing TLBs.

In this new and diverse space, a good design should allow memory systems to scale without being hamstrung by the virtual memory system. This thesis describes the key designs in this space to achieve that goal.

1.2 Our goal

We propose intelligent TLB designs that reduce the address translation overhead significantly so that the memory systems can continue to scale in the presence of the virtual memory system. Our key observation is while operating systems (OS) and hypervisors have a rich set of components in allocating memory, the hardware address translation unit only maintains a rigid and limited view of this ecosystem. Therefore, we seek for patterns inherently present in the memory allocation mechanisms to help catch hardware with the richness of software, thus produce a more intelligent address translation unit.

1.3 Profiling Address Translation Overhead

We have profiled the address translation overhead for some of the key scenarios in Section 1.1. This includes running workloads with large memory footprints on both native as well as virtualized environments. We show our initial findings in Figure 1.2.

Figure 1.2 plots the fraction of application runtime spent on address translation on native and virtualized environments. The evaluated systems use only 4KB page size, which is the most popular pagesize on x86-64. All numbers are collected from on-chip counter measurements. As multi-level TLBs are common in modern processors [1, 2, 5], we show two translation overheads in Figure 1.2, namely the fraction of runtime spent on accessing the second level TLBs after the first level TLB

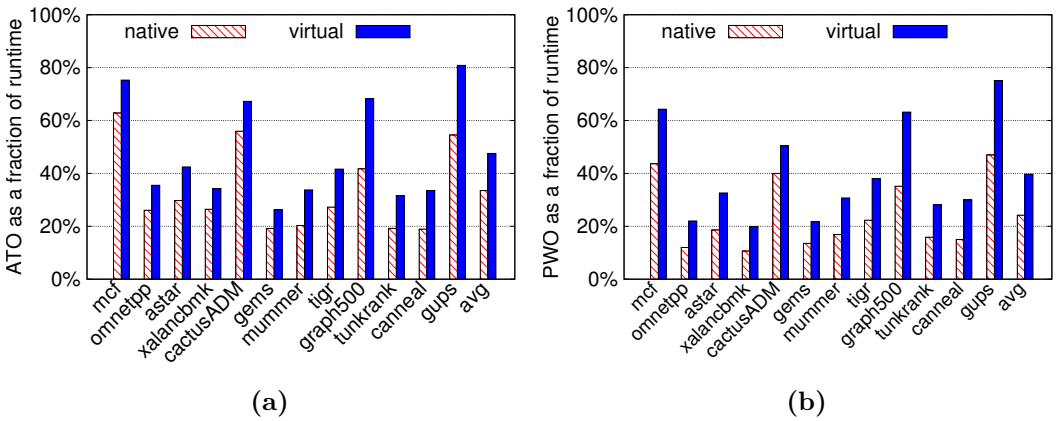


Figure 1.2: (a) Fraction of runtime spent on page table walks; (b) Fraction of runtime spent on looking in the second level TLB and page table walks for 4KB pagesize

misses and walking the page tables after the second level TLB misses in Figure 1.2a, and the fraction of runtime spent on only walking the page tables in Figure 1.2b. Benchmarks from different benchmark suites have been selected, including SPECcpu® 2006 (`mcf, omnetpp, astar, xalancbmk, cactusADM, GemsFDTD`) [39], bioinformatics benchmarks (`mummer, tigr`) [8], graph processing (`graph500, graph analytics`) [32,63], parallel benchmarks (`canneal`) [24], and the RandomAccess benchmark (`gups`) [59]. As can be seen from Figure 1.2, all of the benchmarks have at least 20% translation overhead, and in some particular cases, e.g. `mcf, cactusADM, graph500, gups`, they experience up to 60% translation overhead in native environments. This overhead gets much worse, up to 80% of runtime, in virtualized environments.

One of the potential solutions to mitigate this high overhead is to use larger page sizes, e.g. 2MB in x86 architecture. In general, there are three main benefits of using large pages: lower TLB misses as a single large page is equivalent to 512 small pages, shorter page walk latency as the number of page table levels to be visited is reduced by one, and smaller page table footprint. Figure 1.3b plots the translation overhead when using 2MB page size. As expected, translation overhead is reduced significantly compared to using 4KB page size. However, there are several reasons why large pages are not the universal solution. First, in Figure 1.3b, even though the translation overhead is smaller than in Figure 1.2b, it remains high at 20% on average, and in some particular

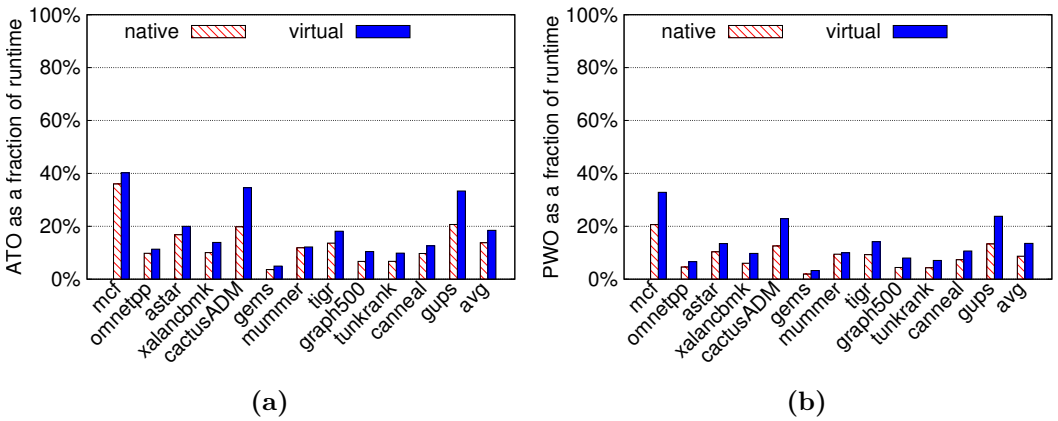


Figure 1.3: (a) Fraction of runtime spent on page table walks; (b) Fraction of runtime spent on looking in the second level TLB and page table walks for 2MB pagesize

benchmarks (e.g. `mcf, cactusADM, gups`), it is close to 40% of system runtime. Second, large pages do not automatically scale with larger physical memory sizes. Third, using large pages conflicts with agile, light-weight memory management mechanisms, e.g. fine-grained protection or page sharing for consolidation. Fourth, if an application's working set is scattered over a wide address space range, large page TLB thrashing can occur [16, 86]. System administrators may therefore disable the hypervisor's ability to back guest large pages [86].

1.4 Dissertation Structure

This dissertation proposes and evaluates several TLB organizations that not only reduce the majority of address translation overhead, but also remain effective in the presence of big memory footprint workloads and multi-dimensional translation process. There are three main chapters:

The two chapters chapter 2 and chapter 3 are strongly related. In these works, we observe that modern operating systems often rely on many software components to manage memory, including buddy allocators, slab allocators, and memory compactors; cumulatively, they allocate as many contiguous (or least spatially-adjacent) physical pages to contiguous (or least-spatially adjacent) virtual pages as possible to reduce memory fragmentation. However, MMUs is not aware of this, hence only associates a

single virtual page to a single physical page. We proposes two novel TLB organizations that maintains an elastic representation of virtual memory. Our designs exploits patterns where many translations exhibit "sequential" or "clustered" spatial locality in which a group or cluster of contiguous or nearby virtual pages map to a similarly contiguous or clustered set of physical pages. As a result, our TLB organizations have higher effective reach than conventional TLBs, and manage to reduce 46% of the page walk overhead, which translates to 7% performance improvement.

The lessons learned from identifying the "sequential" and "clustered" spatial locality help guide us in chapter 4, where we seek for more patterns present in the virtual memory systems, with a focus on address translation in virtual machines. We observe that while large pages are often used to mitigate the high cost of two-dimensional page walks, hypervisors might choose to break them into small pages for better guests memory sharing and monitoring. The tension between large page TLB benefits and fine-grained memory management is regrettable because modern OSes work hard to create large pages. Nevertheless, even when this happens, we see that the majority of these small pages remains well aligned within the boundary of the original large page. Based on this observation, we propose a speculative TLB technique to predict the host physical address directly from the guest virtual address. Our design helps reduce more than 70% of the address translation overhead, which is caused by splintering large pages while running highly consolidated virtualized systems.

Finally, we conclude the thesis in chapter 5.

1.5 Contributions

This dissertation makes the following contributions:

- We have quantified address translation overhead for a range of workloads on both native and virtualized environments, and established that address translation overhead is high in many scenarios and is likely to worsen in the future.

- We have identified and characterized common, yet previously unknown patterns

in page tables that launched a body of work in scalable TLB organizations without explicit hardware-software coordination. This has in turn enabled continued progress in processing and analyzing ever larger datasets for the benefit of a large range of technically, economically, and socially important problems.

Chapter 2

Exploiting Sequential Locality in Page Translations for Large Reach TLBs

2.1 Introduction

In this chapter, we propose techniques exploiting page allocation contiguity that are orthogonal to superpaging, and require no operating system (OS) management overhead. We observe that OS memory allocation mechanisms such as buddy allocators and memory compaction daemons inadvertently allocate contiguous physical page frames to contiguous virtual pages. While these levels of *intermediate contiguity* (in the range of tens of pages) fall short of the contiguity requirements of superpages (hundreds of contiguous pages), they occur naturally and independently of superpages, even in the presence of great system load. In response, we propose Coalesced Large-Reach TLBs (CoLT), a series of hardware mechanisms that allow TLBs to coalesce multiple contiguous virtual-to-physical address translations, thereby enabling them to have greater memory reach. CoLT coalesces multiple virtual-to-physical page translations without explicit OS support or high management overhead. In systems with superpaging, CoLT exploits the vast amounts of contiguity that exist but are insufficient for superpage construction. Even in systems without superpaging, CoLT exploits naturally-occurring contiguity. Our contributions are as follows:

First, we conduct a range of real-system experiments to gauge how often consecutive virtual pages are allocated consecutive physical pages. We find that on Linux, tens of pages are usually contiguous, even in the presence of significant system load and without superpaging support. Furthermore, while superpaging does increase contiguity, most of it falls short of the level necessary to actually create large pages (a 2MB superpage

needs 512 contiguous 4KB base pages, while we see tens of contiguous pages). Instead, we show that TLB coalescing is an effective way of leveraging contiguity.

Second, we propose CoLT using commercially-available two-level TLB hierarchies commonly found in processors today [9,42]. We detail mechanisms to effectively coalesce multiple contiguous virtual-to-physical page translations on set-associative L1 and L2 TLBs, exploring various microarchitectural tradeoffs in their design. Our strategies eliminate roughly 40% of L1 and L2 TLB misses, resulting in average performance improvements of 12%.

Third, we develop CoLT support for small, fully-associative TLBs commonly found in processors to cache superpage entries [9,42]. We show how to overcome the challenge of designing these small structures, achieving L1 and L2 TLB miss elimination rates of 58% on average. These translate to average performance improvements of 14%.

Finally, we explore mechanisms that combine coalescing on both the standard set-associative TLBs and smaller fully-associative TLBs traditionally reserved for superpages. By carefully selecting which structure to allocate coalesced entries in, we achieve L1 and L2 TLB miss eliminations of 55%, yielding average performance improvements of 14%.

Overall, we are the first to observe and exploit intermediate levels of allocation contiguity naturally provided by operating systems. Our techniques are highly effective, yet low overhead. Furthermore, they provide benefits even with heavy system load and regardless of the presence of superpaging.

2.2 Background and Related Work

2.2.1 Prior TLB Enhancement Techniques

Address translation, especially with virtualization and larger application working sets, is a primary source of system performance degradation [19,60]. In response, researchers have considered techniques to improve TLB structure, lookup, and placement [21, 28]. More sophisticated techniques such as TLB prefetching and mechanisms to accelerate page walks have also been considered [14, 23, 48, 77]. While effective, high TLB miss

rates remain problematic.

2.2.2 Superpaging Benefits and Problems

Large pages have been proposed to lower TLB miss rates [31, 33, 64, 74, 83, 84]. Large pages use the same address space as conventional pages but have sizes that are a power-of-two multiple of baseline pages. For example, x86 systems use 4KB baseline pages and support 2MB and 1GB large pages. Furthermore, large pages must be aligned in both virtual and physical memory (superpages of size N must begin at virtual and physical addresses that are multiples of N).

In general, there are three main benefits of using large pages: lower TLB miss rates by replacing hundreds of base page translations with a single superpage translation entry, shorter page walk latency, and smaller page table footprint. However, large pages do not automatically scale up with the larger physical memory. Besides, large pages have high management overheads [64, 83]. They require modifications to OS memory management policies and mechanisms such as support for multiple page sizes and superpage creation. In particular, the process of ensuring that sufficient contiguous physical pages are allocated to virtual pages (to create superpages) has high performance overheads [31,64]. As such, superpages increase the amount of I/O, page initialization/fault latency.

Cumulatively, these problems can easily outweigh the benefits of reduced TLB miss rates. This has a few implications. First, currently-available systems have no universal superpaging support in either the hardware or the OS. Architectures vary in the size of superpages they support (e.g. Intel supports 2MB/1GB superpages while Sparc supports 256MB superpages). Operating systems differ in how they create the contiguity necessary for superpages. For example, FreeBSD uses a reservation-based approach [15], while Linux optimistically allocates 2MB pages, later breaking them into baseline pages if the 2MB version is deemed overly-aggressive [10]. Furthermore, many operating systems do not support superpaging at all or require system administrator intervention to manually construct superpages. Even when superpaging is supported, it is used sparingly to minimize management overheads. In fact, we show that there

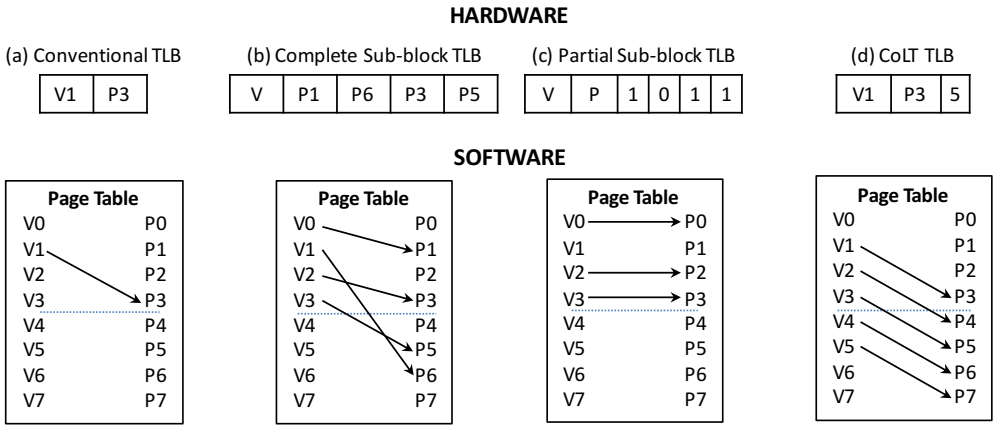


Figure 2.1: Operation of complete sub-blocking and partial sub-blocking versus CoLT TLB. For each approach, we show the structure of a single entry and a page table with the PTEs that can be exploited.

exist immense amounts of page allocation contiguity that remain unharnessed because their level of contiguity is insufficient for superpaging (in the tens of pages rather than the required hundreds of pages).

2.2.3 TLB Subblocking and Speculation

Two hardware-based schemes have been proposed to mitigate superpaging management overheads. Talluri and Hill [83] present *partial-subblock* and *complete-subblock* TLBs, which record ranges of physical pages per virtual page entry, in contrast with the structure of a conventional TLB entry, which only captures a single PTE (in Figure 2.1(a), virtual page V1 and physical page P3)

Complete sub-blocking: Complete sub-blocking looks for clusters of PTEs with contiguous VPNs. For a given sub-block factor N, this approach looks for B *aligned* virtual pages (i.e., all virtual address bits apart from the bottom $\log_2(B)$ bits are the same). It then places all the PTEs corresponding to this group in one complete entry. Figure 2.1(b) shows an example of this where virtual pages 0-3 all are aligned for a sub-block factor of 4. This means that their PTEs can be stored in one entry if it maintains a field for each PPN (e.g., P1, P6, P3, and P5). Unfortunately, the ability of complete subblocking to store any set of PPNs requires expensive hardware (multiple PPN fields).

Furthermore, complete sub-blocking stores a PTE count only equal to the sub-block factor.

Partial sub-blocking: Talluri and Hill proposed partial sub-blocking as a lower-overhead alternative to complete sub-blocking [83]. Figure 2.1(c) shows that partial sub-blocking searches for PTEs with an aligned group of virtual pages *and* an aligned group of physical pages. All PTEs that have VPNs and PPNs with the same offset from the start of the aligned package are coalescable into a single entry. In our example, PTEs for VPNs 0, 2, and 3 achieve this. This approach permits "holes" in a group of PTEs when the physical page offset within the aligned packet is different from the virtual page offset (e.g., the PTE for virtual page 1 in our example). Partial sub-blocking achieves high reach using much simpler hardware than complete sub-blocking by imposing alignment and offset restrictions on PPNs. Figure 2.1(c) shows each entry maintains only a bit vector recording the presence of the physical pages rather than the entire PPN. However, partial sub-blocking's PPN alignment and offset requirements cannot capture many instances of spatial locality. In practice, we find that most instances of spatial locality in PTEs do not fit the alignment requirements of partial sub-blocking (our measurements show that less than 10% of PTEs fit these alignment requirements naturally). While the original partial sub-blocking approach [83] addresses this problem by adding specialized OS code to generate the right alignment and offset features, our goal is to avoid explicit OS modifications.

Alternately, Barr, Cox, and Rixner [15] increase TLB hits with speculation in systems with reservation-based superpaging [64]. Here, physical pages are allocated in aligned 2MB regions of memory corresponding to their alignment within a 2MB region of virtual memory. Since this makes physical page references predictable, Barr, Cox, and Rixner propose SpecTLB, a separate structure to speculatively provide physical addresses when standard TLBs miss. While effective, SpecTLB requires reservation-based superpaging, which is not in widespread use (eg. Linux superpaging [10] does not use reservation-based superpages). SpecTLB also requires additional hardware and can suffer from incorrect speculations.

2.2.4 Our Approach

Our goal is to demonstrate that operating systems naturally generate intermediate levels of page allocation contiguity without explicit support like superpaging, and then to leverage this contiguity to realize large-reach TLBs. Unlike superpaging or subblocked TLBs, we therefore want to avoid OS intrusion and high management overheads. Moreover, we want to exploit *any* amount of contiguity rather than restricting ourselves to set superpage sizes or sizes accommodated by a particular sub-blocked TLB. Unlike SpecTLB, we also do not want to restrict ourselves to one type of superpaging. In fact, CoLT must be effective even *without* superpaging and under high system load. At a high level, as shown in Figure 2.1(d) a CoLT entry maps a group of contiguous, spatially-local PTEs (in our example, PTEs for virtual pages 1-5). Any arbitrary set of PTEs can be accommodated (e.g., five PTEs in Figure **??**) by recording only the base PTE and the number of coalesced PTEs. On look-up, the offset between the base virtual address stored in the tag is used to calculate the offset from the base physical page. There are no alignment restrictions for this approach. Our studies are structured as follows:

We first study how often contiguity exists in real systems, and how this contiguity is affected by buddy allocators, memory compaction superpaging support, and system load. Real-system experiments are particularly crucial since we must establish that contiguity does indeed exist under a variety of scenarios and even in the presence of heavy system fragmentation. Past architectural studies [15] usually focus on simulations to study contiguity; since simulations only capture a small window of the runtime at system boot time when there is little fragmentation, real-system numbers are essential to showcase the wide applicability of our approach.

We then exploit contiguity to realize CoLT, a range of coalesced TLBs. Commercial two-level TLB hierarchies are comprised of set-associative L1 and L2 TLBs, supported by smaller fully-associative buffers to store superpages. We implement coalescing on various combinations of these structures, assessing their benefits and design challenges.

In order to maintain low-overhead designs, unlike past speculation or past prefetching work [15,23,48], we do not augment the standard TLBs with separate structures for our techniques. Therefore, we adjust coalescing designs to seamlessly fit into existing TLB microarchitecture.

2.3 Understanding Page Allocation Contiguity

We now explore why operating systems often allocate contiguous physical page frames to contiguous virtual pages. Since CoLT relies on this behavior, we ascertain which memory allocation policies and mechanisms produce contiguity.

2.3.1 Defining Page Allocation Contiguity

We say that system contiguity exists when consecutive virtual pages are allocated consecutive physical page frames. For example, if virtual pages *1*, *2*, and *3* are allocated physical page frames *58*, *59*, and *60*, we say that these pages are contiguous. Moreover, since this example involves three pages, we say that this is an instance of *3-page* contiguity.

Our definition is distinct from superpages in two ways. First, superpages require a set amount of contiguity. For example, 2MB superpages on x86 systems require instances of *512-page* contiguity. Instead, we make no restrictions on the amount of contiguity that is useful. Second, unlike superpages, we make no assumption on alignment. Overall, our relaxations on contiguity amounts and alignment restrictions reveal huge amounts of *intermediate* contiguity.

2.3.2 Sources of Page Allocation Contiguity

Operating systems maintain a complex set of policies and mechanisms to perform page allocation such that page faults, initialization, and replacement are minimized. A number of these policies have a deep impact on page allocation contiguity. We elaborate on them here, focusing on Linux.

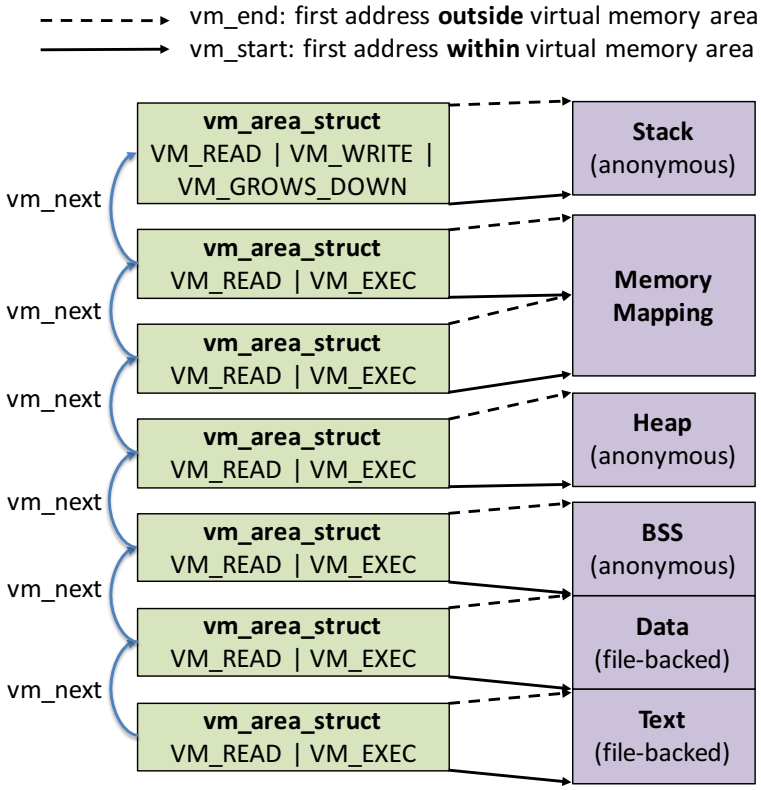


Figure 2.2: Virtual memory areas in process address space.

2.3.2.1 Process address space.

Figure 2.2 shows the layout of the process address space. It consists of multiple virtual memory areas (VMA), which are regions of contiguous virtual addresses, and these regions never overlap. All VMAs are linked together in the form of a red-black tree for fast search operations. Each instance of VMA has several attributes, including its start and end addresses, and access right flags. A VMA that does not map a file is annonymous. Except the memory mapping segment, each memory segment in Figure 2.2, e.g. heap, stack, corresponds to a single VMA.

A VMA is just an agreement between a program and the kernel. When a program allocates memory via a `malloc` or `mmap` call, the kernel just creates or updates the corresponding VMM. The memory request is not actually satisfied until a page fault happens to do real work [37].

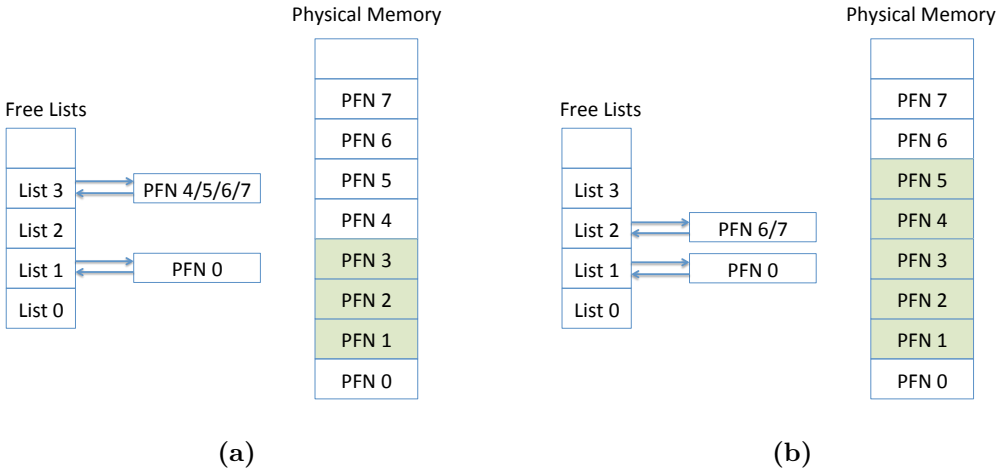


Figure 2.3: (a) Buddy allocator used for physical page allocation. Already allocated pages are shaded, while free pages are tracked by the free lists. (b) Buddy allocator state after an allocation for 2 pages is finished.

2.3.2.2 Buddy allocation.

Most operating systems, including Linux, use a buddy allocator to track physical pages and assign them to virtual pages on demand. Figure 2.3a illustrates the operation of a buddy allocator, assuming that pages 1, 2, and 3 are already allocated. All free physical pages or page frames (PFs) are grouped into ten lists of blocks, which we refer to as *free lists*. Entry x in the free list tracks groups of 2^x *contiguous* physical pages. Since physical page 0 is non-contiguous, it is listed by entry zero. On the other hand, pages 4-7 have 4-page contiguity and are hence listed by entry two.

Physical page allocations proceed as follows. Suppose an application requires an N-page data structure. The buddy allocator first searches the free list entry corresponding to the smallest contiguous page frames bigger than N (entry $\lceil log_2(N) \rceil$). If a block of free physical pages is found in that list, allocation successfully completes. Otherwise, the free list is progressively climbed until an entry with a block of free contiguous physical pages is found. Once a free block is found, the buddy allocator must minimize memory fragmentation. Therefore it iteratively halves the block, inserting these new blocks in their appropriate free list locations, until it extracts a block of N contiguous physical pages. As an example, Figure 2.3b shows the state of the free list after an application level request for two physical pages to be allocated. At first, entry 1 in the

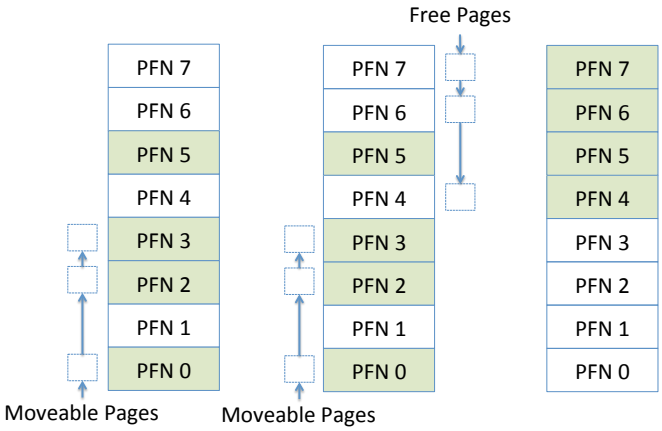


Figure 2.4: The memory compaction daemon tracks movable and free memory pages, exchanging them to eliminate fragmentation.

free list is checked; however, since this is empty, entry 2 is scanned. Here, a free block with contiguous physical pages 4, 5, 6, and 7 is found. Hence, the buddy allocator halves this block of four pages, returning pages 4 and 5 to the application and moving pages 6 and 7 to free list entry 1. Apart from allocation, the buddy allocator also updates its state when physical pages are released. At this point, the kernel attempts to merge pairs of free *buddy* blocks if both have the same size and are contiguous. This merge process is iterative, leading to large swathes of contiguity.

Therefore, though buddy allocation does not produce superpage-level contiguity, it generates large quantities of intermediate contiguity. We will show that the buddy allocator successfully produces this contiguity even in the presence of significant system load. While traditionally irrelevant for TLBs, we aim to exploit this for CoLT.

2.3.2.3 Memory compaction.

While the buddy allocator does try to group together contiguous physical pages on page deallocations, this alone is insufficient to minimize memory fragmentation. Fragmentation is pronounced when multiple processes with large working sets simultaneously run on the system. Therefore, many operating systems, including Linux, boost the buddy allocator with a separate memory compaction daemon. Figure 2.4 details the Linux memory compaction daemon in three steps on a heavily-fragmented system.

First, as shown in the left-most figure, memory compaction runs an algorithm that starts at the bottom of the physical memory and builds a list of allocated pages that are *movable*. While most user-level pages are movable, pinned and kernel pages usually are not. Nevertheless, user-level pages usually outnumber kernel pages, making most pages movable.

Second, the daemon concomitantly runs an algorithm that starts at the top of physical memory and builds a list of free pages. Eventually, the two algorithms meet in the middle of the physical page list. At this point, Linux invokes migration code to shift the movable pages to the free page list, yielding the unfragmented diagram at the right of Figure 2.4.

Since there is a cost associated with moving pages, the compaction daemon is only triggered when there is heavy system fragmentation. As such, its operation naturally produces contiguity, especially in tandem with the buddy allocator. In fact, we will show that this daemon successfully generates contiguity even under heavy system fragmentation.

2.3.2.4 Transparent hugepage support.

Aside from buddy allocation and memory compaction, support for superpages is a primary cause of page allocation contiguity. Unlike those two schemes however, superpage management comes with overheads. As a result, Linux's Transparent Hugepage Support (THS), supported since the 2.6.38 kernel [10], uses superpages carefully. When THS is enabled, the memory allocator attempts to find a free 2MB block of memory. If this block is naturally aligned at a 2MB boundary, a superpage is constructed. In practice, the OS relies on the memory compaction daemon to construct these 2MB regions. When a superpage cannot be constructed, the system defaults to the buddy allocator. Even when the 2MB pages are allocated, increased load can eventually make them harmful. Therefore, system pressure triggers a daemon that breaks superpages into baseline 4KB pages.

We will show that in practice, THS helps create additional levels of contiguity for two reasons. First, if optimistically-allocated 2MB superpages are eventually split due to

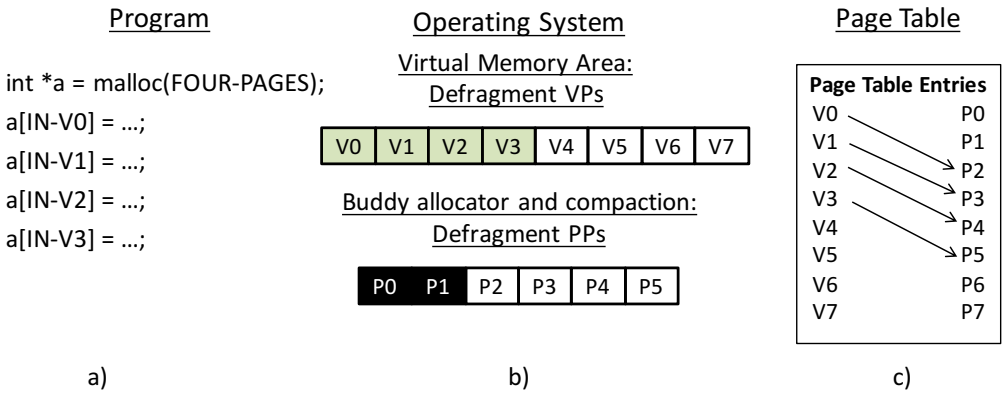


Figure 2.5: Interaction between program's fault order and memory management.

system pressure, they retain contiguity among tens of baseline 4KB pages. Second, THS relies on the memory compaction daemon, triggering it more often and providing the buddy allocator even higher levels of contiguity. Therefore, though THS uses superpages sparingly, it constructs intermediate contiguity useful for CoLT.

2.3.2.5 Putting Things Together.

In this section, let's see how everything works together to generate the intermediate contiguity between virtual pages and contiguous physical pages. Figure 2.5 shows the interaction between program faults order and kernel's memory management. In this example, a program tries to allocate four memory pages in Figure 2.5(a). As mentioned in Section 2.3.2.1, this request is not honored right away by the kernel. Instead, the kernel simply updates the heap VMA and returns – V0 to V3 region in Figure 2.5(b). No page frames are actually allocated and the new pages are not present in physical memory. Once the program actually accesses the pages, the processor page faults and invokes a page fault handler in the kernel. The handler searches for the VMA covering the faulted virtual address, when a VMA is found, the kernel creates a PTE to map the faulted virtual page to a allocated physical frame.

With the help from the buddy allocator and memory compaction, region of free contiguous physical pages have been formed, P0 to P3 in Figure 2.5(b). If program faults in order, as shown in Figure 2.5(a), physical frames will be allocated in order

from the contiguous physical pool. As a result, consecutive mappings between virtual and physical pages are established.

2.3.2.6 System Load and Memory Fragmentation.

Finally, page allocation contiguity is deeply affected by the system load. If many processes run simultaneously, main memory is likelier to be fragmented. Therefore, one may initially expect that higher load degrades contiguity. Surprisingly, we will show that contiguity can actually *increase* with greater system load. This occurs because system load has a complex relationship with the memory compaction daemon, triggering it more often when there is higher load. This can, in turn, provide greater swathes of contiguous physical frames to the buddy allocator, eventually resulting in more contiguity.

2.4 CoLT Design and Implementation

Having detailed contiguity sources, we now propose three variants of CoLT. Overall, they share three design principles. First, they detect instances of consecutive virtual-to-physical address translations. These entries are coalesced into single TLB entries, so as to reduce miss rates. Second, CoLT coalesces only on *TLB misses*. While TLB hits could also prompt coalescing, this may increase lookup latencies. Third, coalescing is *unintrusive*, unlike speculation and prefetching [15, 23, 48, 77] which can degrade performance. For example, incorrect speculations suffer a high penalty. Incorrect prefetches lead to the eviction of useful entries and higher bandwidth usage. Prior work mitigates these problems by TLB speculating or prefetching using separate structures [15, 23, 48]. In contrast, we coalesce entries directly into the TLBs but ensure that coalescing occurs only around *on-demand* translations. In the worst case, coalesced entries may be unused but are not harmful. This is crucial given that system contiguity does not necessarily imply that all contiguous translations are used in temporal proximity. We ensure coalesced entries are available if needed but do not harm TLB hit rates when they are unused.

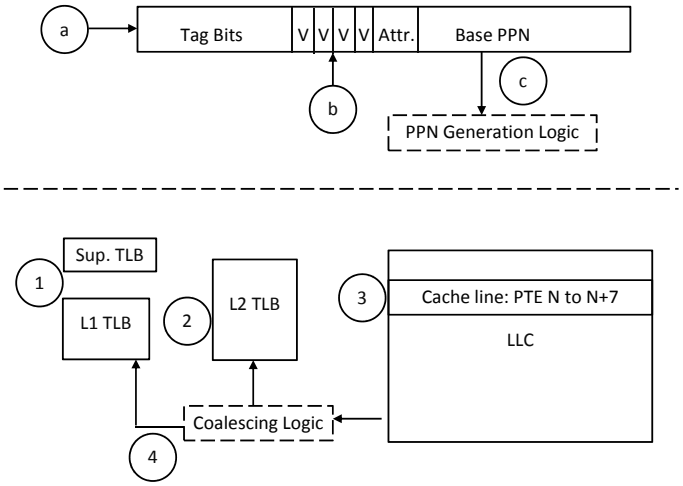


Figure 2.6: CoLT for set-associative L1 and L2 TLBs.

We propose three variants of CoLT for commercial two-level TLB hierarchies. This hierarchy contains set-associative L1 TLB and L2 TLBs, used to cache baseline 4KB pages [9, 42]. Superpages are cached in separate small, fully-associative TLBs that are accessed in parallel with the L1 TLB. Note that the L2 TLB is inclusive of just the set-associative L1 TLB and not the superpage TLB.

There are three natural coalescing mechanisms for this hierarchy. First, we coalesce in just the set-associative L1 and L2 TLBs. Second, we coalesce in the superpage TLB only. Third, we use a combined approach that routes some coalesced entries to the set-associative TLBs and others to the superpage TLBs. We now describe each of these schemes.

2.4.1 CoLT-SA Design and Implementation

CoLT-Set Associative (CoLT-SA) coalesces multiple virtual-to-physical page translations in the set-associative L1 and L2 TLBs. We first detail its high-level operation and then focus on specific design challenges.

2.4.1.1 Overall operation.

The bottom half of Figure 2.6 shows a high-level view of CoLT-SA. In step 1, the set-associative L1 TLB and superpage TLB are looked up in parallel. Assuming L1 and L2

TLB misses (step 2), a page table walk brings in the desired translation entry into the LLC (step 3). At this point, two parallel events occur. First, the requested translation is returned to the processor pipeline. In parallel, the *Coalescing Logic* studies the translations around the requested entry for contiguity. It coalesces as many of these translations as possible, as long as map to the same set. This entry is inserted into the L1 and L2 TLBs (step 4). As we will detail, conventional set-associative TLBs map consecutive virtual addresses (and hence contiguous translations) to consecutive sets, precluding coalescing. We therefore modify the virtual page bits used for set-selection so that translations for groups of consecutive virtual page numbers do map to the same set, allowing for potential coalescing. Furthermore, since we provide the requested translation to the pipeline in parallel with the *Coalescing Logic's* operation, the latter is off the critical path and does not affect TLB miss handling times.

2.4.1.2 TLB set selection.

Conventional set-associative TLBs place translations of successive virtual page numbers into successive sets, preventing coalescing. In response, we modify TLB set selection. For example, a TLB with 8 sets uses bits 2 to 0 of the virtual page number for set selection (VPN[2-0]). Instead, we left-shift the index bits by $log_2(N)$ bits if we want to place N consecutive translations in the same set (permitting us to coalesce a maximum of N translations per entry). In our example, to ensure that translations with four consecutive virtual pages map to the same set, we use VPN[4-2] as the new indexing bits.

To coalesce more entries, the indexing bits are further left-shifted (for example, to coalesce up to eight entries, VPN[5-3] must be used). However, using higher order bits for set indexing increases conflict misses since more consecutive entries are mapped to the same set. This is a fundamental tradeoff for CoLT-SA designs – in choosing the correct index bits, we must balance opportunities for coalescing with potentially higher conflict misses. We find that allowing for coalescing of four contiguous translations generally performs best.

2.4.1.3 Lookup operation.

The top half of Figure 2.6 illustrates CoLT-SA lookups. Each coalesced TLB entry maintains tag bits, the higher order bits left of the index bits used for set selection. For example, if up to four contiguous translations can be coalesced in a TLB with eight sets, VPN[4-2] is used for set selection and VPN[63-5] is the tag. In step (a), this tag is checked against the requested virtual page number. In step (b), the non-index lower-order virtual page bits (VPN[1-0] in our example) are used to select among multiple valid bits. There is one valid bit for every possible translation in a coalesced entry. These valid bits indicate the presence of a translation in the coalesced entry. If on step (b), a valid bit is set, there is a TLB hit. At this point, extra logic calculates the physical page number. CoLT entries store the base physical page number for each coalesced entry. This number corresponds to the virtual page represented by the first set valid bit. Therefore to reconstruct the physical page number, combinational logic (*PPN Generation Logic*) calculates the number of valid bits away this entry is from the first set valid bit. This number is then added to the stored base physical page number to yield the desired physical page.

We believe this lookup operation remains low-overhead and will not impact TLB access cycle times. First, the initial tag match and check of valid bits is simple. The PPN generation logic addition is also low-overhead as the amount of coalescing is bounded (in our example, at best, an addition of four will be required). As such, readily-implementable combinational logic, similar to logic used to calculate prefetching strides and addresses or update branch predictor state, can calculate the physical page number. This is substantially lower-overhead than prior prefetching schemes requiring dedicated adders [48]. This combinational logic will not affect TLB access cycles, adding just a few gate delays at best.

2.4.1.4 Practical coalescing restrictions.

Ideally, after the page table is walked to handle a TLB miss, coalescing logic finds as many contiguous translations around the requested translation as possible. Practically,

however, coalescing is restricted by two constraints. First, as we have already discussed, the choice of index bits for set selection places a limit on coalescing opportunity. A second limit arises from our desire to minimize the overhead associated with searching for contiguous translations. On a TLB miss, a page table walk finds the desired translation. We aim to prevent any additional page walks when checking for contiguous entries adjacent to the requested translation. Since the page table walk accesses the last-level cache (LLC) and brings data in 64-byte cache line sizes, seven additional translations are fetched. These translations are brought without additional memory references; thus we check just them for contiguity. In practice, this approach restricts coalescing to a maximum of eight translations. Despite this restriction, CoLT eliminates a high number of TLB misses.

2.4.1.5 Replacement, invalidations, and attribute changes.

CoLT-SA assumes standard LRU replacement policies. While there may be benefits in prioritizing entries with different coalescing amounts differently, we leave this for future work. We also assume a single set of attribute bits for all the coalesced entries. This restricts the amount of contiguity that we can exploit; more sophisticated schemes supporting separate attribute bits per translation in a coalesced entry will improve our results. Furthermore on TLB invalidations, we flush out entire coalesced entries, losing information for pages that would be unaffected in standard TLBs. Gracefully uncoalescing TLB entries and only invalidating victim translations will perform even better. This too is the subject of future work.

2.4.2 CoLT-FA Design and Implementation

Rather than supporting coalescing in set-associative TLBs and changing their indexing scheme, we can instead coalesce into just the fully-associative TLB. This structure is usually used exclusively for superpages. However, superpages are often used sparingly. Hence, allocating coalesced entries into this structure in addition to superpages entries may be beneficial. We refer to this as CoLT-Fully Associative (CoLT-FA).

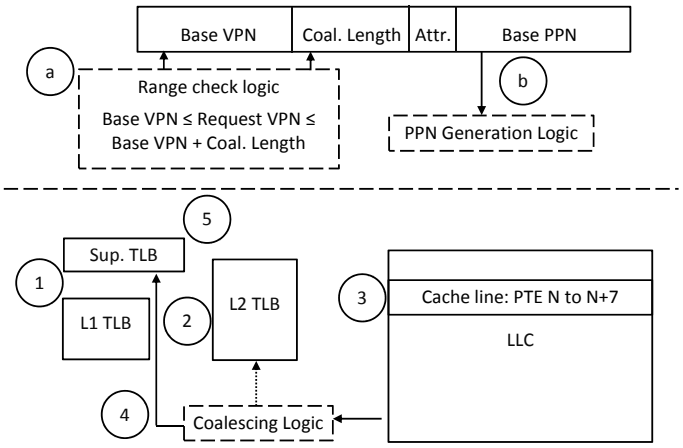


Figure 2.7: CoLT for the fully-associative superpage TLB.

2.4.2.1 Overall operation.

The bottom half of Figure 2.7 delineates CoLT-FA operation. Assuming misses in all the TLBs (steps 1 and 2), a page walk is conducted in step 3. At this point, a cache line provides up to eight translations that can be checked for contiguity. Up to eight translations are now coalesced in step 4. If coalescable, the entry is loaded into the fully-associative TLB. If no coalescing is possible, it is loaded into the set-associative L1 and L2 TLBs.

On insertion into the fully-associative TLB, further coalescing is possible. Since contiguity may exist between the newly coalesced entry and a resident entry in the fully-associative TLB, the latter is scanned for a coalescing check. As the requested translation can be sent to the pipeline in parallel, this scan is off the critical path and does not impact miss handling times. If possible, further coalescing is done in step 5.

The main benefit of this scheme is that the set-associative TLBs remain unaffected and can now be devoted to non-contiguous translations while the superpage-TLB captures all contiguous translations. Empirically, however, we have found that due to the small size of the superpage-TLB, useful entries are frequently evicted. Therefore, for performance reasons, when bringing a coalesced entry into the fully-associative structure, we still bring just the requested entry (and not its coalesced neighbors) into the L2 TLB. While this does create some redundancy in terms of stored entries, we will show

that performance is improved. Note that we leave the L1 TLB unaffected due to its much smaller capacity. Note also that CoLT-FA shares both superpage entries and coalesced entries in a single structure. One initial concern may be that if coalesced entries far outnumber superpage entries, the latter will be evicted from the fully-associative TLB. In practice, we find that this is not a problem for two reasons. First, superpages are used sparingly, requiring a very small number of entries in the buffer. Second, when used, these superpages are frequently accessed, meaning that they remain at the head of the LRU list, preventing their eviction.

2.4.2.2 Lookup operation.

The top half of Figure 2.7 details CoLT-FA lookup. Each coalesced entry maintains a base virtual page number as the tag and a field that logs the number of entries coalesced. Unlike CoLT-SA, there are no coalescing restrictions due to indexing schemes. We find that using 10 bits for the coalescing length field suffices as this captures a contiguity of 1024 pages. Since a TLB entry uses 64-bytes, this overhead is minimal. Each entry also stores the base physical page and attributes of all contiguous translations.

In step (a), *Range Checking Logic* compares the requested virtual page number against the range of translations stored by each entry of the fully-associative TLB. As shown, comparator and adder logic is required for the range check. If the virtual page is detected in the range, there is a TLB hit. At this point (step (b)), the *PPN Generation Logic* subtracts the tag base virtual page number from the requested virtual page number. This value is then added to the stored base physical page number to find the desired physical page number.

While the hardware required for range checks and physical page number generation is more complex than CoLT-SA, it can still be accomplished using purely combinational logic. We nevertheless stray on the conservative side in our experiments by reducing the size of the fully-associative TLB in CoLT-FA as compared to the baseline case without coalescing. Commercial systems tend to implement 16 to 24-entry [42] fully-associative TLBs for superpages. To ensure that the added lookup complexity does not bias our results, we assume only 8-entry fully-associative TLBs with coalescing. This ensures

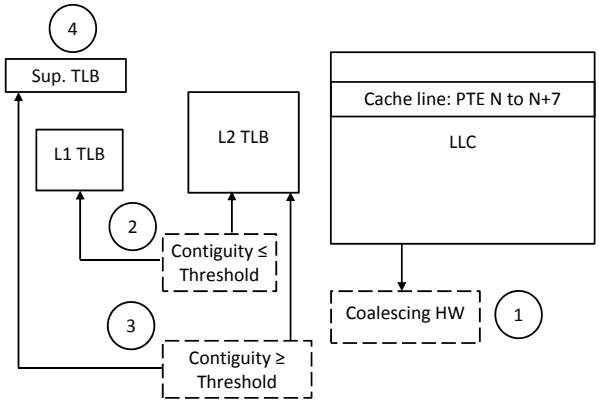


Figure 2.8: Combined CoLT for all TLBs.

that the total time to check the TLB for a hit definitely does not increase. Even with this overly-conservative assumption, we find that CoLT-FA is effective.

2.4.2.3 Replacement, invalidations, and attribute changes.

We assume standard LRU for the fully-associative structure. Due to its smaller size, we suspect though that smarter replacement policies will be even more effective. Furthermore, we share the same attribute bits for all coalesced entries and invalidate entire entries, but for larger amounts of coalescing. Despite this, we will show that CoLT-FA performs effectively.

2.4.3 CoLT-All Design and Implementation

Finally, CoLT-All coalesces into both set-associative L1/L2 TLBs and the superpage TLB. Its primary benefit over CoLT-SA and CoLT-FA is that it provides potentially the largest reach, at the expense of modifying both the set-associative and superpage TLB. For large-data applications though, the benefits will far exceed the hardware modifications required.

2.4.3.1 Overall operation.

Figure 2.8 illustrates CoLT-All's operation when all the TLBs experience a miss. In step 1, the page walk has occurred and the coalescing hardware has determined the

amount of contiguity present in the cache line. In then checks this contiguity to see how it compares to a threshold. If it is lower than a threshold (step 2), this means that the contiguity can be accommodated by the indexing scheme of the set-associative TLBs. For example, suppose the contiguity is three pages and we use an 8-set TLB with VPN[4-2] for indexing (allowing coalescing of up to four translations). In this case, the coalesced entry is allocated into the set-associative L1 and L2 TLBs. However the contiguity may be higher than the threshold and the amount that the set-associative TLBs can accommodate. In our example, the contiguity may be five. In this case, the entry is coalesced and brought into the superpage-TLB. At the same time, because the superpage-TLB is small, useful coalesced entries may be frequently evicted. Therefore, like CoLT-FA, we allocate an entry at this point into the L2 TLB as well. Unlike CoLT-FA however, our set-associative L2 TLB can now also handle coalesced entries (albeit with smaller levels of coalescing permissible by its choice of index bits). Therefore, CoLT-All brings in as much of this coalesced entry as possible into the L2 TLB, unlike CoLT-FA which brings just the requested translation. Finally, in step 4, the new allocated superpage entry may be coalesced with already-resident entries.

2.4.3.2 Lookup operation.

Lookup operates similarly to CoLT-SA and CoLT-FA. On a memory reference, both the L1 and the superpage TLB are checked. If an entry is found in either, it is raised to the top of its LRU list. Therefore, both TLBs now have support to generate the physical address.

2.4.3.3 Replacement, invalidation, and attribute changes.

There are no changes in the replacement, invalidation, and attribute policies of CoLT-All from CoLT-SA and CoLT-FA.

2.5 Methodology

We now detail the infrastructure and workloads used to quantify real-system contiguity and CoLT's effectiveness at leveraging this contiguity to eliminate TLB misses. Our analysis focuses on data pages since data references cause far more misses than instruction references [22, 77].

2.5.1 Real-System Characterizations of Page Allocation Contiguity

2.5.1.1 Experimental platform and methodology.

We use a system with a 64-bit Intel i7 processor, 64-entry L1 TLBs, and a 512-entry L2 TLB, a 32KB L1 cache, a 64KB L2 cache, a 4MB last-level cache (LLC), and 3GB of main memory. Furthermore, we run Fedora 15 (Linux 2.3.68).

To measure contiguity, we hack the kernel to scan the page table looking for instances of contiguous address translations. We walk the page table every five seconds, capturing contiguity changes through the benchmark run. Our original definition of contiguity is based only on page numbers; however, we now additionally require that contiguous translations must share the same page attributes and flags. While this eases the hardware implementation of CoLT by allowing for the same set of attribute bits per coalesced entry, contiguity would be even higher if this constraint were relaxed.

To study the effect of memory compaction, we use the Linux `defrag` flag. Enabling this flag triggers the memory compaction daemon both on page faults and as system background activity. Disabling this flag greatly reduces the number of times the memory compaction daemon runs. In tandem, we enable and disable THS to study the impact of superpaging. We also ensure that our system is realistically fragmented by using a machine that has already run a number of applications (eg. web browsers, network clients, office utilities) for two months. To further load the system, we run `memhog`, a memory fragmentation utility [25], with our workloads. We study scenarios where `memhog` fragments 25% and 50% (a highly fragmented system when combined with the other background activities) of the memory. In all, we thus study system twelve configurations. Due to space constraints, this paper focuses on the following specific

Benchmark	Suite	THS on L1/L2 MPMI	THS off L1/L2 MPMI
Mcf	Spec	56550/28600	95600/49230
Tigr	BioB.	19000/18150	26950/18860
Mummer	BioB.	12910/11450	14760/12970
CactusADM	Spec	6610/8140	8420/6930
Astar	Spec	8480/4660	17390/11240
Omnetpp	Spec	8410/2730	34040/8080
Xalancbmk	Spec	2670/2150	14120/2100
Povray	Spec	7010/630	7310/630
GemsFDTD	Spec	1300/620	8030/3620
Gobmk	Spec	710/410	1550/510
FastaProt	BioB.	460/300	610/300
Sjeng	Spec	1840/200	3860/440
Bzip2	Spec	4070/150	7120/270
Milc	Spec	120/90	3780/1820

Figure 2.9: Summary of benchmarks used in our studies.

ones:

1. THS on, normal memory compaction, no memhog: this is the current default setting for Linux.

2. THS off, normal memory compaction, no memhog: this shows contiguity *without* superpaging.

3. THS off, low memory compaction, no memhog: conservative case for contiguity because neither THS no memory compaction occur. The buddy allocator struggles to find contiguous physical blocks.

4. THS on, normal memory compaction, memhog: we test the effect of system load on the default Linux setting by assigning 25% and 50% of system memory to memhog.

5. THS off, normal memory compaction, memhog: shows the impact of fragmentation without superpaging.

Our contiguity studies go beyond past work [14, 15, 21], which simulate systems just after boot time, without fragmentation. Our results are therefore more comprehensive and realistic.

2.5.1.2 Evaluation workloads.

We study system contiguity on the SPECcpu® 2006 benchmarks [39] and bioinformatics workloads from Biobench [8] in Table 2.9. We run each of the workloads with

their maximum data sets (for SPECcpu, this corresponds to *Ref*) to completion. From the real-system runs, we use on-chip performance counters to track L1 and L2 TLB misses per million instructions (MPMI) when THS support is enabled and disabled. The benchmarks are ordered from highest to lowest THS on L2 TLB MPMIs. `Mcf`, `Tigr`, `Mummer`, `CactusADM`, and `Astar` see particularly high TLB MPMIs. While enabling superpaging does reduce TLB misses for some workloads, it alone is insufficient. For example, `Mcf` still has an L2 TLB MPMI of 57K with THS on, while `Mummer` is unchanged.

2.5.2 Simulation-Based CoLT Evaluations

2.5.2.1 Simulated system.

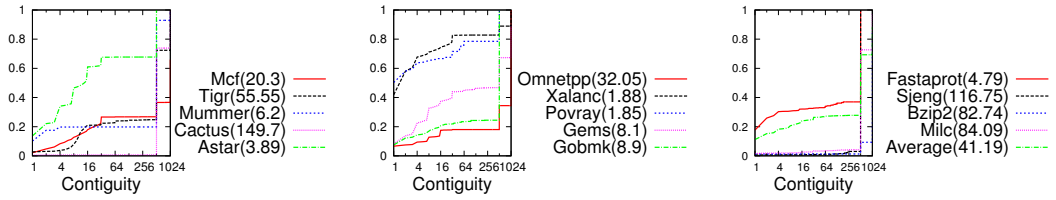
Past work on TLBs [14,15,21,77] focuses on miss rates rather than performance because it is infeasible to run memory-intensive applications for long enough duration to provide practical performance numbers. While we do study miss rates, we go beyond previous work by considering the performance impact of our approaches. We use a two-step evaluation to quantify changes in hit rate and to then offer performance numbers feasible for simulations.

Like the bulk of recent work on TLB analysis, we first use a trace-based approach to analyze miss rates [14, 15, 21, 22]. We extract detailed memory traces by simulating an x86 processor on Simics [85]. These highly detailed traces maintain logs of both data and instruction references at the micro-op level. Our traces also capture full-system effects by running benchmarks on a Linux 2.6.38 kernel. We hack the simulated kernel to provide full page table walk details for every single memory reference (this includes the virtual page, the physical page, and all attribute bits). We set the kernel to its default configuration of using THS and normal memory compaction. As we will show, since contiguity is present across all kernel configurations, CoLT will be effective across the range of superpaging and memory compaction settings.

We run the traces through a highly-detailed custom memory simulator. We need to stress our TLBs using simulated workloads in a manner that matches real-system stress;

therefore, we use 32-entry and 128-entry L1 and L2 4-way set-associative TLBs. These sizes are chosen as they produce simulated load within 10% of the load experienced by a real system. Our baseline system also assumes a 16-entry fully-associative superpage TLB. As previously detailed, CoLT-FA and CoLT-All reduce this size to 8 entries in order to provide conservative performance improvement data and negate the impact of slightly more complex lookups. Furthermore, unlike past work [21, 23], we model a more realistic TLB hierarchy with 22-entry MMU caches, accessed on TLB misses to accelerate page table walks [14]. Finally, we assume a three-level cache hierarchy similar to the Intel Core i7 (32KB L1 cache, 64KB L2 cache, 4MB LLC).

Having assessed miss rates, we now go beyond prior work and the study the performance implications of our approach. We use the Pin-based [57] CMP$im [45] simulation framework to model a 4-way out-of-order processor with a 128-entry reorder buffer. The processor's TLB and cache parameters match those of our custom trace module. Unfortunately, the simulation speeds of this detailed microarchitectural framework are slow; hence we cannot use it to run full Linux distributions with the memory allocation behavior necessary to study CoLT on sufficiently long-running, large-data applications. However, we can use this infrastructure to gauge the overall runtime overheads expended on TLB misses. In tandem with the miss rate eliminations extracted from our trace-based approach, this allows us to interpolate CoLT's actual performance gains. This interpolation strategy is valid for two reasons. First, TLB miss penalties (page walks) are serialized as only one page walk can typically be handled at a time [21, 23]. Hence, TLB misses lie on the execution's critical path. Second, our interpolation approach is actually conservative as it does not account for the instruction replays that would likely occur on TLB misses. Therefore, our projected performance benefits would likely *increase* on a real system. The bottomline is that our simulation approach combines the feasibility of miss rate simulation of prior approaches with *more detailed performance insights than prior work.*



(a) **(b)** **(c)**

Figure 2.10: THS on, normal memory compaction contiguity CDF.

2.5.2.2 Evaluation workloads.

We use the workloads from Table 2.9 for our evaluations. However, due to slow simulation speeds, we use Simpoints [67] that total to one billion instructions per workload. These simpoints include operating system effects captured by Simics and assume realistic inputs (for SPECcpu, this corresponds to *Ref*).

2.6 Real-System Characterizations of Page Allocation Contiguity

We now quantify how the buddy allocator, memory compaction, THS, and system load affect application contiguity on a real system. We show that page allocation contiguity *always* exists regardless of the kernel configuration.

We begin by discussing the cumulative density functions (CDFs) from Figure 2.10 to Figure 2.12. These show the distribution of contiguities experienced by non-superpage pages. Note that contiguity (the x-axis) is presented as a log scale.

2.6.1 Superpaging, Memory Compaction

Figure 2.10a, Figure 2.10b, and Figure 2.10c, ordering the benchmarks from highest to lowest TLB MPMI, show contiguity assuming default Linux kernel settings (superpaging and normal memory compaction). The legend provides average contiguity numbers.

Figure 2.10 shows that there is heavy contiguity across the workloads that cannot be exploited by superpages. On average, pages are in 41-contiguity groupings. Furthermore, the CDFs show that there can be large instances of contiguity above the average. For example, most CDFs see many 64 to 256-contiguity instances.

Interestingly, there exist many cases of 512 and 1024-page contiguity. Since THS is

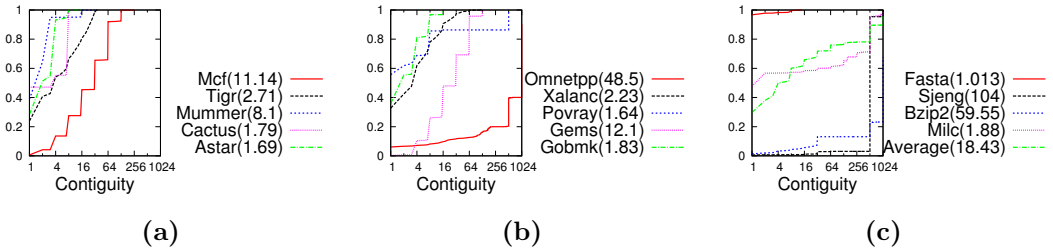


Figure 2.11: THS off, normal memory compaction contiguity CDF.

enabled, one might initially expect that these should be treated as superpages. However, this contiguity does not translate to superpages for two reasons. First, these memory chunks are not superpage-aligned. Second, THS currently supports superpaging for only *anonymous* pages created through `malloc` calls; as such, file-backed pages created from `mmap` calls are not superpage candidates. Overall, we find that 15% of non-superpage pages actually have over 512-page contiguity.

Fortunately, Figure 2.10 enjoys particularly high contiguity for TLB-stressing benchmarks. `Mcf`, `Tigr`, and `CactusADM` see tens to hundreds of contiguous pages, indicating their amenability to TLB coalescing. For a number of these benchmarks, such as `Mcf`, high contiguity arises because `malloc` and `mmap` calls are made at the beginning of the execution to allocate large hash-based data structures. These structures span megabytes of space, which the buddy allocator ensures maps to contiguous physical pages.

2.6.2 No Superpaging, Memory Compaction

Figure 2.11a to Figure 2.11c show how contiguity changes when superpaging support is disabled. Average contiguity drops compared to THS on from 41 to 18, for two reasons. First, THS optimistically creates as many 2MB page as possible. While these 2MB pages eventually get broken into 4KB pages due to system load, they do leave large amounts of smaller, residual contiguity. Without THS, contiguity is not generated this way. Second, disabling THS drastically reduces memory compaction daemon invocations. Nevertheless, sufficient exploitable intermediate contiguity remains (in the tens of pages, around 18). Furthermore, heavy TLB-pressure benchmarks like `Mcf` and `Mummer` see very high contiguity.

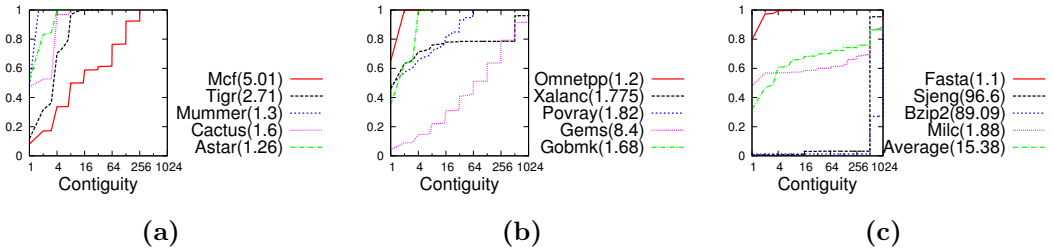


Figure 2.12: THS off, low memory compaction contiguity CDF.

Surprisingly, some benchmarks like `Omnetpp` and `Sjeng` actually see *higher* contiguity without THS. This occurs because the lack of THS reduces superpages allocated to *other* running processes. As a result, the pages allocated to our workloads remain unfragmented and contiguous.

2.6.2.1 No Superaging, Low Memory Compaction

Figure 2.12a, Figure 2.12b, and Figure 2.12c present a worst-case scenario setting for Linux, where THS is turned off and memory compaction is greatly reduced via disabling the `defrag` kernel flag. While *no* kernel uses or recommends this setting, we study it to ensure that sufficient contiguity exists, even when there are almost no mechanisms to explicitly generate it. In fact, our results show that on average, contiguity drops only marginally compared to the THS off, normal memory compaction case to 15 pages on average. While important benchmarks like `Mcf`, `Mummer`, and `Omnetpp` do lose compared to the prior settings, they retain sufficiently high intermediate contiguity. For example, even though `Mummer's` average contiguity is now 1.3, roughly 50% of its 4KB pages enjoy 4-page contiguity. Correctly exploiting this gives our TLBs a 4× reach.

2.6.3 Superpaging, Memory Compaction, Memhog

We now focus on the impact of system load on fragmentation and contiguity. Figure 2.13 shows how contiguity is affected when `memhog` runs with each benchmark and fragments 25% and 50% of system memory. Our studies have shown that combined with the other running system processes, `memhog` with 50% heavily fragments almost all memory and causes page fault rates to greatly increase. We assume default Linux settings (THS

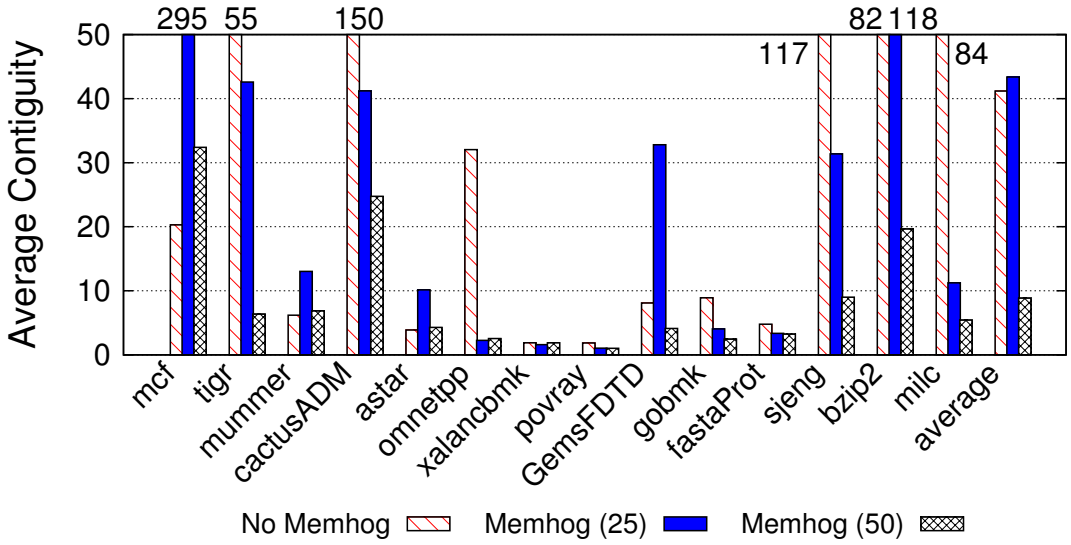


Figure 2.13: Average contiguity for THS on, normal memory compaction with varying `Memhog`.

enabled, normal memory compaction).

One might initially expect higher system load to result in lower contiguity. Surprisingly, we find the *opposite* trend to hold when using `memhog(25%)`, with contiguity rising from 41 to roughly 43 pages on average. For some benchmarks, the gain is markedly high; for example, `Mcf` and `GemsFDTFD` contiguities are boosted by an order of magnitude. The primary reason for this is that higher load invokes the memory compaction daemon more often. This in turn provides the buddy allocator more contiguous physical blocks. This bodes particularly since coalescing contiguous translations will likely become *more* effective under system load.

Greatly fragmenting the system with `memhog(50%)` however, does reduce contiguity. However, even this intermediate contiguity is relatively high, averaging close to 10 pages. For heavy TLB-pressure benchmarks like `Mcf` and `Mummer`, this configuration still achieves higher contiguity than without system load. As such, the buddy allocator, in tandem with memory compaction, manages to actually leverage the additional load to increase contiguity.

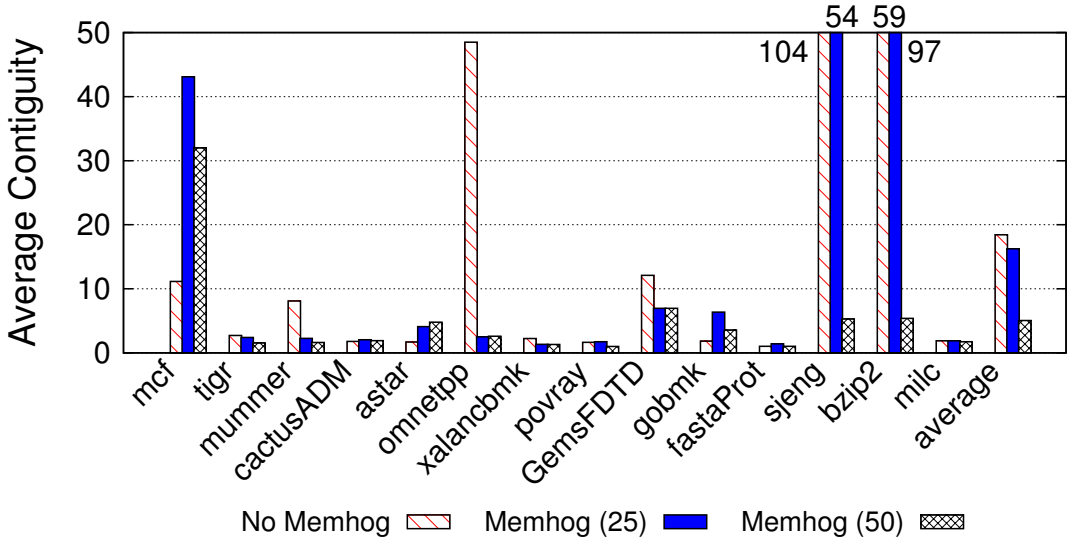


Figure 2.14: Average contiguity for THS off, normal memory compaction with varying `Memhog`.

2.6.4 No Superpaging, Memory Compaction, Memhog

This represents the scenario where THS is turned off despite high system load. While kernel settings would not typically allow this, we use this setting to stress-test our measurements. We find that even under the pessimistic setting of no THS and `memhog(50%)`, the average contiguity is about 5. TLB coalescing can thus potentially provide a 5× reach.

2.6.5 Summary of results.

Three primary conclusions can be drawn from our real-system characterizations. First, under *every* single configuration, even those that are unrealistically severe, the buddy allocator, compaction daemon, and THS support succeed in inadvertently generating great intermediate contiguity. Second, system load can have surprising implications on contiguity, often increasing it. For some benchmarks that suffer from high TLB misses, such as `Mcf`, this is a promising observation. Third, superpages are ill-equipped to handle this contiguity. Therefore, coalescing techniques to harness this intermediate contiguity is warranted.

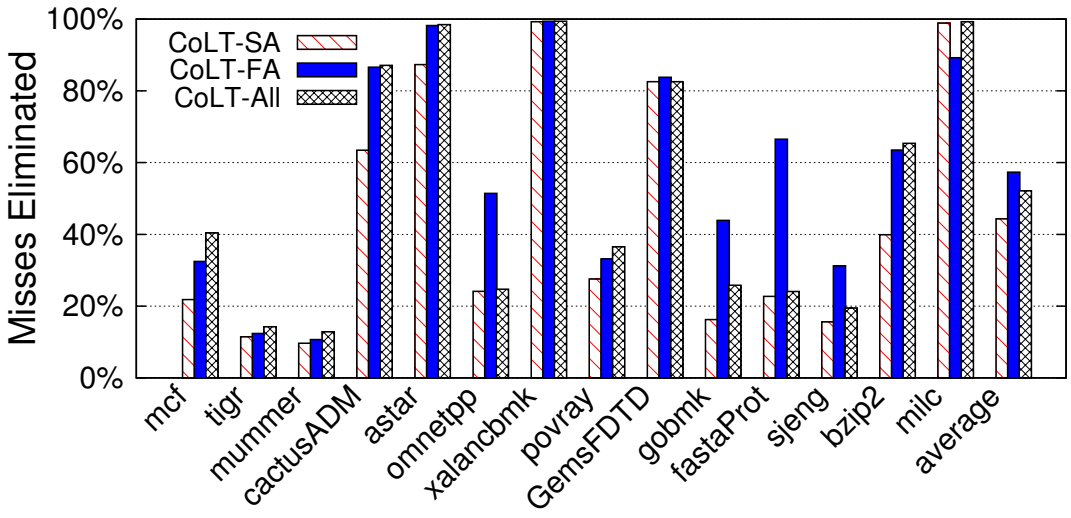


Figure 2.15: Percentage of L1 TLB misses eliminated using CoLT-SA, CoLT-FA, and CoLT-All normalized to baseline TLB misses.

2.7 CoLT Evaluations

We now evaluate CoLT's benefits, focusing on per-application miss rate reductions and performance gains. We begin by showcasing how CoLT-SA, CoLT-FA, and CoLT-All eliminate both L1 and L2 TLB misses.

2.7.1 TLB Miss Rate Analysis

2.7.1.1 CoLT TLB miss rates.

Figure 2.15 and Figure 2.16 quantify CoLT's TLB miss reductions. The benchmarks are ordered from highest to lowest TLB miss rates. For every benchmark, we first capture the number of L1 and L2 TLB misses for a baseline configuration with 32-entry and 128-entry L1 and L2 TLBs (4-way) and a 16-entry superpage TLB. Note that we count misses for both the set-associative L1 TLB and the superpage TLB as L1 TLB misses since they are checked in parallel and have the same hit time. After recording these misses, we then run the same benchmarks on configurations with CoLT-SA, CoLT-FA, and CoLT-All, tracking the new TLB miss rates. We assume that CoLT-SA uses VPN[4-2] and VPN[6-2] for L1 and L2 set selection, meaning that up to four translations can be coalesced per entry (we will later show the effect of using more aggressive indexing).

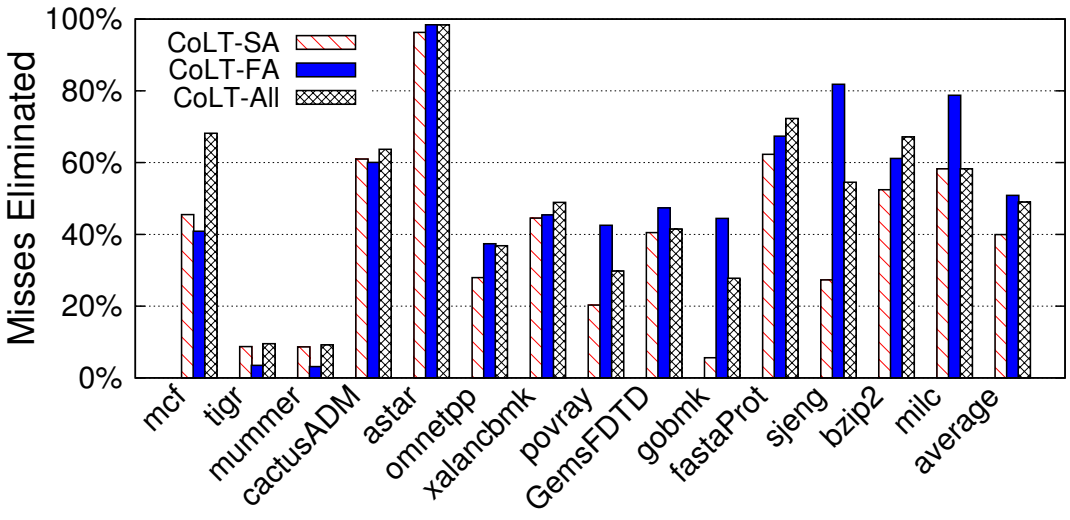


Figure 2.16: Percentage of L2 TLB misses eliminated using CoLT-SA, CoLT-FA, and CoLT-All normalized to baseline TLB misses.

We also conservatively assume 8-entry fully-associative TLBs when using CoLT-FA and CoLT-All.

First and foremost, Figure 2.15 and Figure 2.16 show that all three CoLT schemes improve *every* single benchmark by eliminating large chunks of the baseline misses. On average, CoLT-SA eliminates 40% of both L1 and L2 TLBs misses, while CoLT-FA and CoLT-All do even better, eliminating around 55% of both L1 and L2 misses. These large miss eliminations imply great performance potential. Second, Figure 2.15 and Figure 2.16 show that many of the benchmarks experiencing TLB pressure gain particularly from CoLT. For example, `Mcf`, `CactusADM`, and `Astar` all eliminate vast amounts of their TLB misses. In fact, `Astar` almost achieves perfect TLBs with no misses with CoLT-FA and CoLT-All.

Third, there is a general correlation between the amount of system contiguity and effectiveness of CoLT. For example, `Mcf`, `Bzip2`, `Milc`, and `CactusADM`, which all see more instances of 20-page contiguity on average, can coalesce large amounts of translations, increasing TLB reach substantially. However, contiguity alone does not guarantee coalescing success; for coalescing to be effective, contiguous entries must actually be used close together in time. Without this temporal proximity, a coalesced entry will be evicted from the TLB before multiple member translations are used. This explains the

lower benefits of `Tigr`, which sees 10% TLB miss elimination rates despite a contiguity of over 50 pages on average.

Fourth, Figure 2.15 and Figure 2.16 clearly show that leveraging the superpage TLB in CoLT-FA and CoLT-All provides 10-15% gains over CoLT-SA on average. We see that benchmarks like `Mcf` and `Fastaprot` benefit particularly from this. These gains are achieved *despite* dropping from a 16-entry to an 8-entry structure. We find that the primary reason for this is that even with THS on, superpages are used sparingly. Therefore, a surprisingly high number of entries remain wasted in the fully-associative TLB in the baseline case. Instead, CoLT-FA and CoLT-All use these entries and can even perform unrestricted coalescing on them, unlike the set-associative TLBs.

The difference between CoLT-FA and CoLT-All remains more nuanced. We find generally that both schemes eliminate roughly 55% of TLB misses on average. Generally on the more pressured benchmarks (eg. `Mcf`, `Tigr`, `Mummer`, `CactusADM`), CoLT-All outperforms CoLT-FA slightly. This is because both sets of TLBs can leverage coalescing in this case, increasing overall effective TLB coverage.

Overall, all CoLT designs eliminate a large fraction of TLB misses. We now focus on implementation details of the various CoLT designs to lend greater insight on our gains.

2.7.1.2 Impact of CoLT-SA's indexing scheme on TLB miss rates.

Our initial CoLT-SA results assume that we use VPN[4-2] and VPN[6-2] for L1 and L2 set selection. This limits the amount of coalescing to four translations per entry. While additional contiguity could be coalesced by further left-shifting the index bits, this also increases conflict misses. Figure 2.17 and Figure 2.18 study these opposing forces on the 4-way associative L1 and L2 TLBs respectively by left-shifting the traditional index bits by one bit (VPN[3-1] and VPN[5-1] for L1 and L2 TLBs), two bits, and three bits (VPN[5-3] and VPN[7-3] for L1 and L2 TLBs). These correspond to maximum allowable coalescing of two, four and eight translations. We do not left-shift beyond this since we allow coalescing only for translations on a single cache line (therefore, a maximum of eight entries may be compressed).

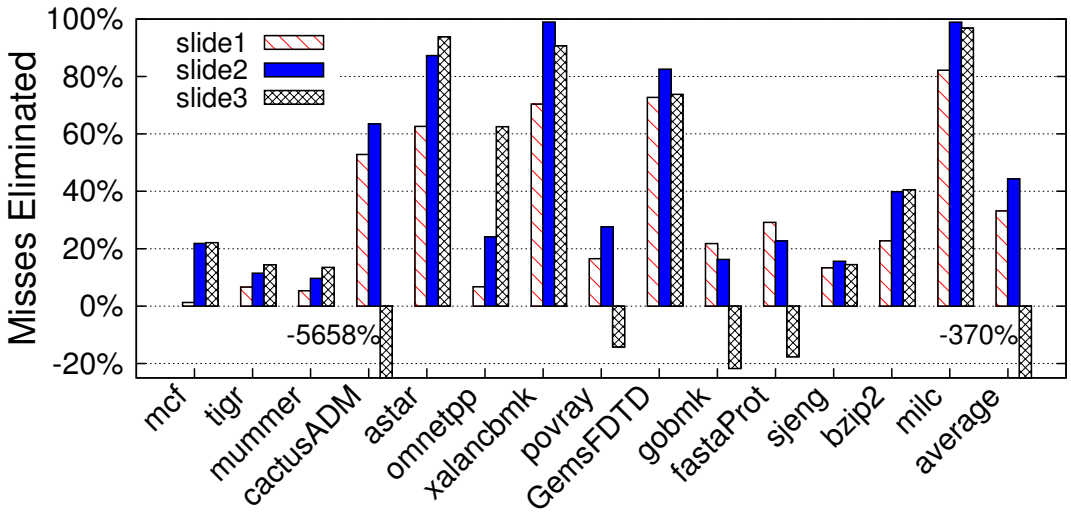


Figure 2.17: Effect of left-shifting index on L1 misses.

Figure 2.17 and Figure 2.18 clearly show that left-shifting the index bits by two provides the best balance between coalescing opportunity and conflict misses. Below this (left-shifting by one bit), we can only coalesce two entries, restricting our TLB miss elimination rates. However, left-shifting by three bits actually *increases* TLB misses in many cases due to the additional conflict misses. In general, unless there is very high contiguity, like for `Mummer`, `Tigr`, and `Milc`, left-shifting the index bits by three is overly-aggressive. Given these results, we assume a left-shit of two bits for our indexing scheme.

2.7.1.3 Impact of bringing missing entries into L2 TLB for CoLT-FA and CoLT-All.

As previously detailed, while CoLT-FA and CoLT-All bring coalesced entries into the fully-associative, superpage TLB, they also leverage the L2 TLB. For CoLT-FA, when a coalesced entry is brought into the superpage TLB, just the requested entry is also brought into the L2 TLB; for CoLT-All, a coalesced entry (where the coalescing amount is restricted by the index scheme) is brought into the L2 TLB. As we previously noted, this is useful since the superpage TLB is small (8-entry); as a result, only entries with high levels of coalescing are maintained there. As such, intermediate-level coalesced entries are often evicted. Bringing these entries into the L2 TLB as well increases the

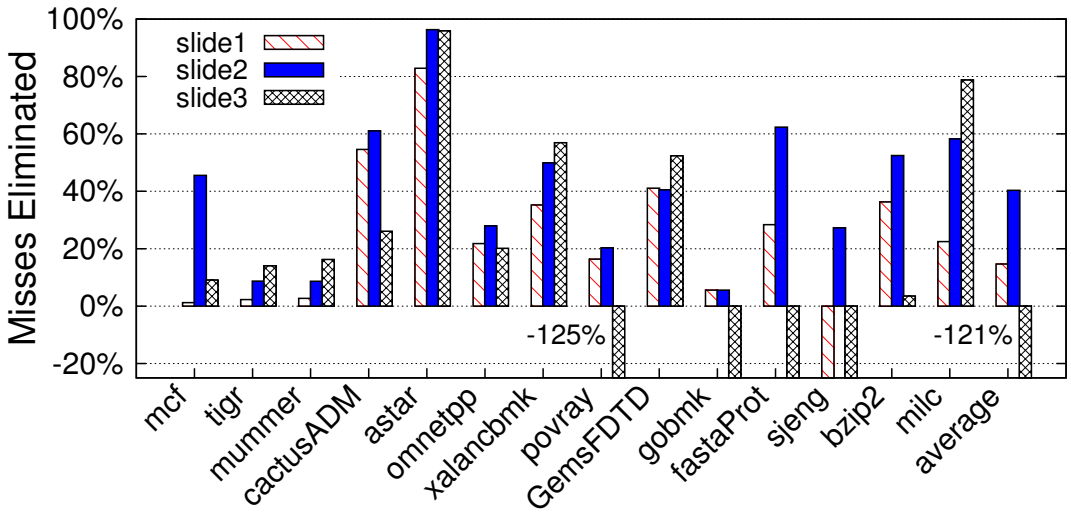


Figure 2.18: Effect of left-shifting index on L2 misses.

chance that these they remain available if necessary.

For CoLT-FA, we have run experiments to compare the case when (1) a coalesced entry is brought into the superpage TLB and just the translation triggering the coalescing is also brought into the L2 TLB, and (2) a coalesced entry is brought into the superpage TLB but the L2 TLB remains unaffected. We have found that on average, (1) outperforms (2) by an additional miss elimination of 10-15% for both L1 and L2 miss counts. We see particularly high gains with our approach on workloads with relatively lower contiguity such as `Povray`, since the small superpage TLB cannot coalesce a high enough number of entries to prevent eviction.

For CoLT-All, we have similarly run experiments to compare the case when (1) a coalesced entry is brought into the superpage TLB and its smaller coalesced version (a maximum of coalescing of four translations in our design) is brought into the L2 TLB, and (2) a coalesced entry is brought into the superpage TLB but the L2 TLB remains unaffected. We see again that our approach, (1) outperforms (2) by an average of 10-20% TLB miss eliminations.

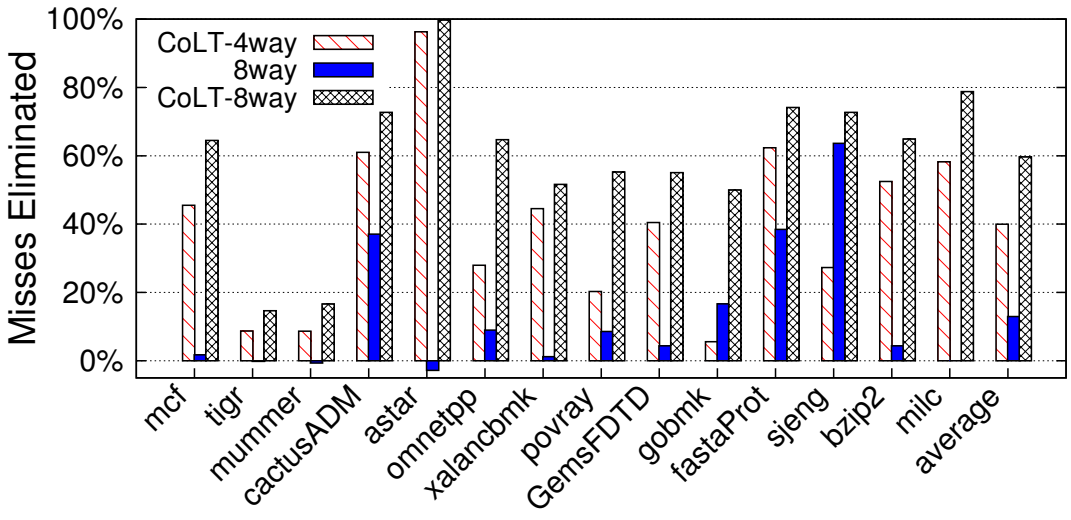


Figure 2.19: Percentage of baseline misses eliminated by CoLT-SA when increasing associativity.

2.7.1.4 Studying CoLT's effectiveness at higher associativities.

We now consider CoLT effectiveness as TLB associativity is varied. A number of past studies have quantified how effectively increasing TLB associativity eliminates misses [28]. Generally, these studies have concluded that the slightly higher TLB hit rates are offset by huge power dissipation problems [17]. These observations are largely responsible for the relatively low associativity (typically 2-way or 4-way) supported on current TLBs.

CoLT, however, increases the benefits of higher set-associativity since the indexing scheme of the TLB can be more aggressively changed without as significant an increase in conflict misses. Overall, this allows higher levels of coalescing. Figure 2.19 compares how many L2 TLB misses in a 4-way 128-entry L2 TLB can be eliminated by CoLT-SA (4-way, CoLT-SA), by varying the associativity to 8-way but not allowing coalescing (8-way, No CoLT), and by allowing CoLT on the 8-way TLB (8-way, CoLT-SA). Note that all configurations use a fixed TLB size despite associativity changes.

First, Figure 2.19 shows that merely increasing the associativity to 8-way only eliminates 10% of the baseline L2 misses. In fact, even 4-way L2 TLBs with low-overhead CoLT-SA far exceed the benefits of higher associativity, eliminating 40% of baseline misses on average.

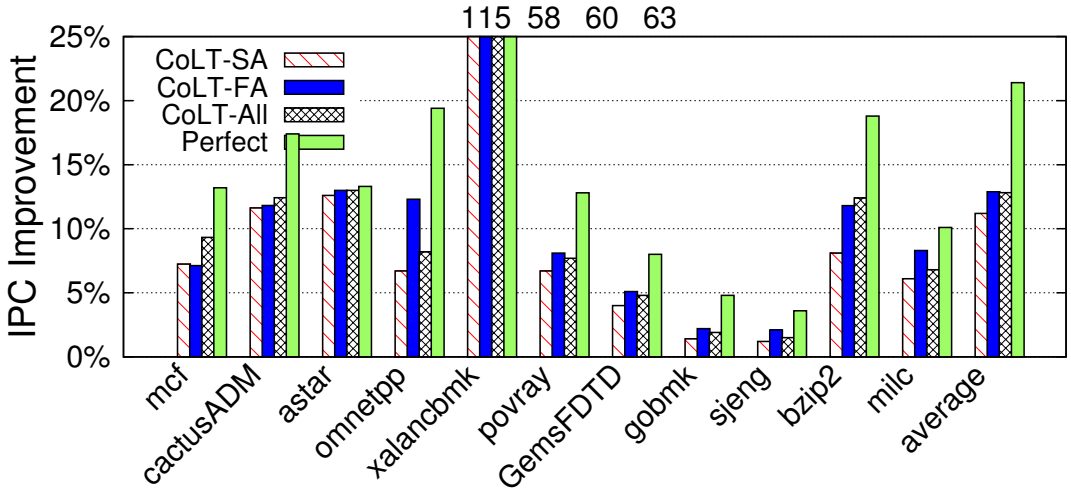


Figure 2.20: CoLT-SA, CoLT-FA, and CoLT-All performance improvements compared to perfect TLBs with 100% hit rates.

Figure 2.19 shows however, that the 8-way configuration, if augmented with CoLT-SA, does provide significant benefits. CoLT-SA now eliminates 60% of the baseline misses on average, a substantial improvement over the other two scenarios. While a detailed power analysis is beyond the scope of this work, the performance to power ratio may therefore become more amenable with CoLT. We leave a detailed analysis of this scheme as the subject of future work.

2.7.2 Performance Analysis

Up to this point, we have evaluated the benefits of CoLT in terms of miss rate eliminations. While this does indicate CoLT's effectiveness, we now focus on performance numbers which track how much faster each application runs with coalescing. Figure 2.20 details, for every benchmark, performance improvements from CoLT-SA, CoLT-FA, and CoLT-All. It also provides data on performance improvements that would occur with absolutely perfect, 100%-hit rate TLBs. The latter serves as a comparison point to determine how effectively CoLT performs. Once again, the baseline is a system with 4-way 32-entry and 128-entry L1 and L2 TLBs, and a 16-entry superpage TLB. CoLT-FA and CoLT-All conservatively reduce the superpage TLBs to 8 entries. Moreover, as previously detailed, we simulate a 4-way out-of-order processor. Note that results are

shown for only the Spec 2006 benchmarks because our infrastructure cannot currently support the Biobench workloads.

Figure 2.20 shows that perfect TLBs would improve most benchmark runtimes by over 10% (eg. all except `Gobmk` and `Sjeng` in our benchmarks). In fact, `Xalancbmk` sees a huge 115% improvement in performance from TLBs that achieve 100% hit rate. These numbers indicate that TLB miss handling does significantly slow down benchmarks. This also implies that CoLT strategies have the potential to significantly improve performance.

Fortunately, Figure 2.20 shows that all of the CoLT approaches do indeed boost application performance significantly. On average, CoLT-SA achieves a 12% performance improvement, while CoLT-FA and CoLT-All achieve 14% improvements. On benchmarks like `Xalancbmk`, the performance improvements hover around 60% of runtime. Across other workloads like `Mcf`, `CactusADM`, `Astar`, `Omnetpp`, and `Bzip2`, at least one of the CoLT configurations improves performance over 10%. Already substantial, we anticipate that as applications with even larger working sets or virtualization are considered, these performance improvements will be even higher.

Interestingly, Figure 2.20 indicates that CoLT-SA, which has arguably the simplest implementation, performs almost as well as CoLT-FA and CoLT-All on average. Nevertheless, some benchmarks like `Omnetpp` and `Bzip2` do see large boosts from CoLT-FA/CoLT-All over CoLT-SA. Because our results do assume smaller 8-entry fully-associative TLBs, we expect CoLT-FA and CoLT-All results to be even higher with more realistically-sized superpage TLBs.

2.8 Summary

In this chapter, we propose and design Coalesced Large-Reach TLBs capable of exploiting address translation contiguity to achieve high reach. Due to a variety of OS memory management techniques involving buddy allocators, memory compaction, and superpaging, large amounts of translation contiguity are generated, even under heavy system load. While this contiguity typically cannot be exploited to generate superpages,

CoLT provides lightweight hardware support to detect this behavior. As a result, large TLB miss eliminations are possible (on average, 40% to 58%), translating to performance improvements of 14% on average.

Chapter 3

Exploiting Clustered Locality in Page Translations for Large Reach TLBs

3.1 Introduction

Chapter 2 has proposed coalesced large-reach TLBs (CoLT) based on the observation that, independent of large pages, operating systems typically exhibit "contiguous" spatial locality (although this is not guaranteed) in which tens of consecutive virtual pages are mapped to consecutive physical pages. This behavior, caused in part by OS buddy allocators and memory compaction, generates many instances of contiguous PTE spatial locality, though typically not enough for large page generation. CoLT proposes novel but modest hardware changes to a conventional TLB to exploit contiguous spatial locality.

In this chapter, we go beyond CoLT by observing that many translations exhibit "clustered" spatial locality in which translations are "nearby" in the same address region. We provide a detailed characterization of PTE spatial locality across many workloads. We use both detailed simulations and real-system approaches and show that weakly-clustered spatial locality is more prevalent than contiguous spatial locality. Next, we propose a low-overhead, multi-granular TLB organization that exploits PTE clustering. Our approach uses modest hardware and no OS support, making it robust for applications ranging from the server and desktop to high-performance computing and cloud-computing domains. We consider enhancements to our design (e.g., replacement policies and prefetching) that eliminate 46% of L2 TLB misses on average. Overall, our approach largely subsumes the prior CoLT technique by exploiting contiguous spatial locality when it exists, therefore it is thus effective even when OSs have been running

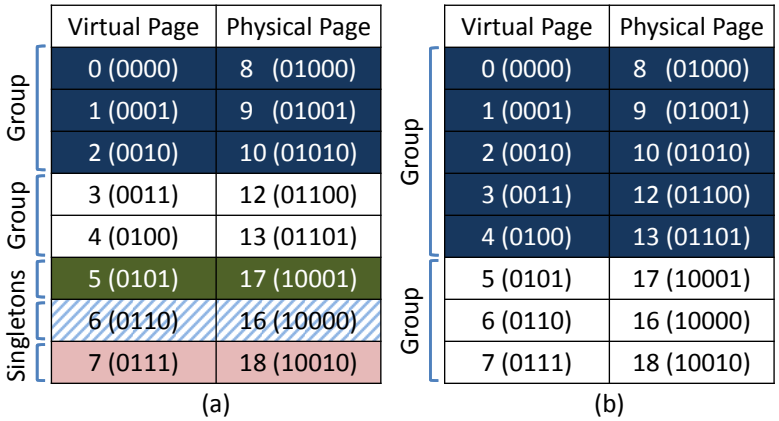


Figure 3.1: The figure on the left shows the presence of contiguous spatial locality (sequential groups) in a page table. The figure on the right shows that if clustered locality is also observed, the entire page table can be more efficiently covered.

for long periods and are fragmented, making contiguous spatial locality harder to find.

3.2 Related Work and Our Approach

3.2.1 Spatial Locality in Page Table Entries

Large pages exploit cases in which large swathes of contiguous virtual pages are assigned contiguous physical pages. We refer to groups of adjacent PTEs as *contiguously spatially local*. Large pages require explicit OS intervention to ensure ample contiguous spatial locality; however, it is also possible for operating systems to generate intermediate amounts of spatial locality (in the range of tens to a few hundreds of PTEs). For example, Figure 3.1(a) shows a page table in which the PTEs for virtual page numbers (VPNs) 0-2 are in a sequential group of physical pages. Similarly, the sequential group of PTEs for VPNs 3-4 are contiguously spatially local. Chapter 2 shows that this behavior occurs often even in the absence of large page support because of OS buddy allocators and memory-compaction daemons. Overall, we find that our page table is made up of two sequential groups of PTEs exhibiting contiguous spatial locality and three additional "singleton" PTEs.

This chapter's key insight is that there exists another form of spatial locality, likely occurring in greater abundance than contiguous spatial locality. Specifically, we find that many PTEs exhibit *clustered spatial locality* in which a cluster of nearby virtual

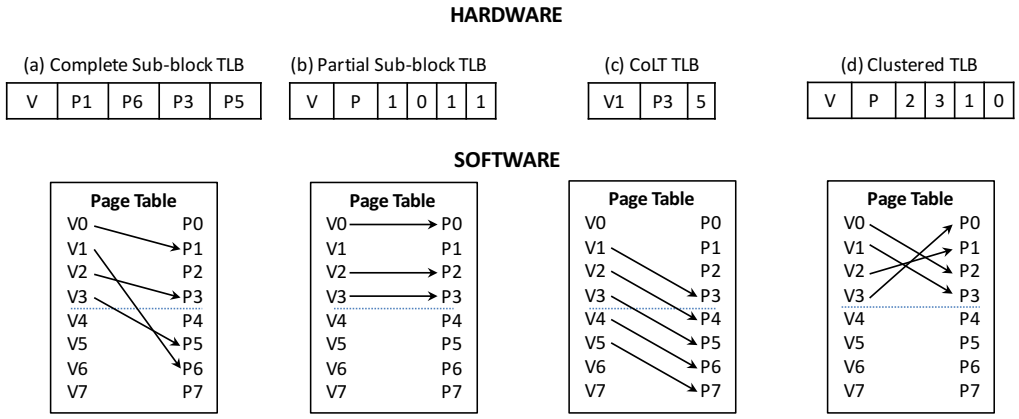


Figure 3.2: Operation complete sub-blocking, partial sub-blocking, and CoLT versus clustered TLB. For each approach, we show the structure of a single entry and a page table with the PTEs that can be exploited.

pages map to a similarly clustered set of physical pages. Consider Figure 3.1(b), assuming that we scan for clusters of up to eight PTEs. In our example, PTEs demonstrate clustered spatial locality if they all share the same VPN divided by 8 *and* the same physical page number (PPN) divided by 8 (i.e., we ignore the bottom 3 VPN and PPN bits). Therefore, the entire page table is covered by two clusters. The first cluster matches the first two sequential group of PTEs from Figure 3.1(a), and the second cluster comprises PTEs for VPNs 5-7. The goal of our work is to show that this form of clustered locality is abundant, even in fragmented systems, and to design low-overhead hardware to exploit these patterns.

3.2.2 Other Techniques to Exploit Page Table Spatial Locality

In chapter 2, we have discussed the differences between sub-blocking and CoLT. In Figure 3.2, we compare and contrast these previous approaches with clustered TLB.

Coalesced Large-reach TLBs (CoLT): Recall that a CoLT entry maps a group of contiguous, spatially-local PTEs (in Figure 3.2(c), PTEs for virtual pages 1-5). Any arbitrary set of PTEs can be accommodated (e.g., five PTEs in Figure 3.2(c)) by recording only the base PTE and the number of coalesced PTEs, and there are no alignment restrictions for this approach. CoLT achieves high reach but is entirely reliant on contiguous

spatial locality.

Complete sub-blocking: This approach, shown in Figure 3.2(a), relaxes the need for contiguous spatial locality [83]. Instead, complete sub-blocking looks for clusters of PTEs with contiguous VPNs. Unfortunately, the ability of complete sub-blocking to store any set of PPNs requires expensive hardware (multiple PPN fields). Furthermore, unlike CoLT which accommodates any length of contiguous PTEs, complete sub-blocking stores a PTE count equal to the sub-block factor.

Partial sub-blocking: This is a lower-overhead alternative to complete sub-blocking, shown in Figure 3.2(d). Partial sub-blocking searches for PTEs with an aligned group of virtual pages *and* an aligned group of physical pages. This approach permits "holes" in a group of PTEs when the physical page offset within the aligned packet is different from the virtual page offset (e.g., the PTE for virtual page 1 in our example). Partial sub-blocking achieves high reach using much simpler hardware than complete sub-blocking by imposing alignment and offset restrictions on PPNs.

Intuitively, partial sub-blocking goes beyond CoLT by exploiting contiguous spatial locality *and* limited forms of clustered locality. However, its PPN alignment and offset requirements cannot capture many instances of clustered spatial locality (e.g., the third cluster in Figure 3.1 cannot be leveraged because it requires VPNs 5 and 6 to map to PPNs 16 and 17, respectively, to be useful). In practice, we find that most instances of clustered spatial locality in PTEs do not fit the alignment requirements of partial sub-blocking (our measurements show that less than 10% of PTEs fit these alignment requirements naturally). While the original partial sub-blocking approach [83] addresses this problem by adding specialized OS code to generate the right alignment and offset features, our goal is to avoid explicit OS modifications.

3.2.3 Our Approach: Clustered TLBs

We design a multi-granular TLB architecture that exploits more general forms of spatial locality. We focus on clustered PTE spatial locality that also largely subsumes contiguous spatial locality. We achieve this using a novel clustered TLB architecture.

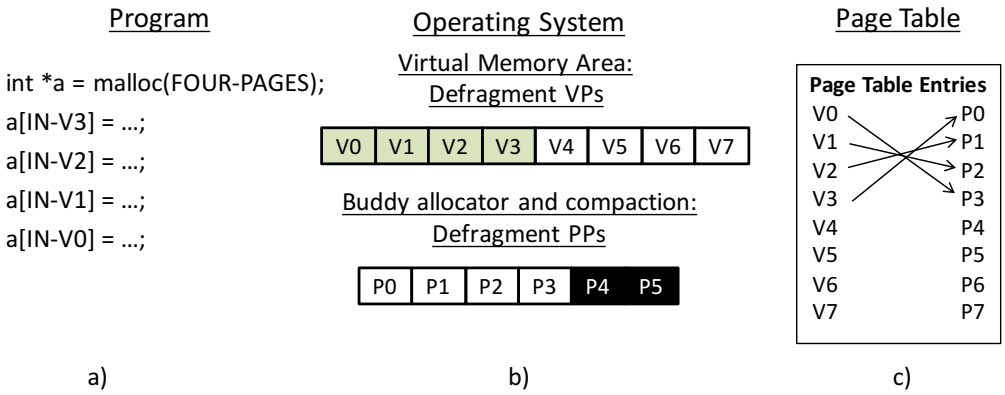


Figure 3.3: Interactions between program faults order and kernel's memory management.

Figure 3.2(d) shows a clustered TLB. Like sub-blocked TLBs, a clustered TLB is designed for a maximum cluster factor of N (in this example, N=4). Suppose an aligned group of virtual pages is found; if these virtual pages also point to an aligned group of physical pages, the PTEs can be placed in a single clustered TLB entry. This means that: (1) all VPNs in a cluster share the same bits ignoring the bottom $\log_2(N)$ bits; and (2) all PPNs in a cluster share the same bits ignoring the bottom $\log_2(N)$ bits. Unlike partial sub-blocking, these PPNs have no additional offset requirements in a cluster (in our example, V0 maps to P2, but V2 to P1) and permit holes. An N-wide vector is maintained per entry to record the offset of the physical pages in the aligned group (only the $\log_2(N)$ bottom-order bits are needed for this). In this way, clustered TLBs can capture *all* the clusters from Figure 3.1.

Figure 3.3 shows the interaction between program faults order and different memory management mechanism in the kernel, namely VMAs, buddy allocator, and memory compaction. As discussed in Section 2.3.2.5, buddy allocator and memory compaction help create areas of contiguous physical addresses to be returned to the program. Difference from the example in Figure 2.5, this program accesses pages in reverse order, where it touches page V3 first, followed by V2, V1 and V0. As a result, reverse mappings are established in the page tables in Figure 3.3(c). Reverse faulting is just one of many faulting types that cannot be captured by the CoLT design, and we try to exploit here with clustered TLB.

Our approach offers key advantages relative to past work. Unlike CoLT, it captures clustered spatial locality, making it robust even in the presence of fragmentation. Second, unlike sub-blocking, it uses much simpler hardware and does not require explicit OS support. Subsequent sections detail our design. We characterize the presence of clustered locality across a variety of workloads and system configurations. We then show how a multi-granular design made up of a clustered TLB coupled with a small conventional TLB effectively captures this clustered spatial locality.

3.3 Weak Spatial Locality in Page Tables

In this section, we characterize spatial locality in page tables. We analyzed a total of eleven benchmark traces from SPEC workloads (xalancbmk, SPECweb, GemsFDTD, astar, omnetpp, mcf), server workloads (TPC-C, Trade6), and CloudSuite workloads [32] (Graph Analytics, Data Serving, Data Caching)[1]. These traces were correlated with hardware performance counters to ensure that they exhibit similar behaviors. We first characterize the opportunities for previously-proposed CoLT-like approaches, and then we relax the constraints on how PTE entries may be coalesced and examine the impact that has on the potential for coalescing.

3.3.1 CoLT-like Contiguous Spatial Locality

We measured the fraction of PTEs that exhibit CoLT-styled contiguous spatial locality, in which contiguous virtual pages map to contiguous physical pages. Each graph in Figure 3.4 corresponds to one application. The x-axis shows the number of PTEs that can be grouped, the y-axis is percentage of all PTEs, and the graphs show cumulative distributions. For example, the bold solid line (labeled "Contiguous" in the legend) for omnetpp shows that 46% of all PTEs have a grouping size of one (i.e., they cannot be coalesced with any adjacent PTEs), about 96% of translations can be coalesced into groups of six or fewer consecutive PTEs, and all PTEs can be coalesced into groups consisting of no more than eight consecutive PTEs. Across the benchmarks, there are some

[1] See Section 4.6 for methodology details.

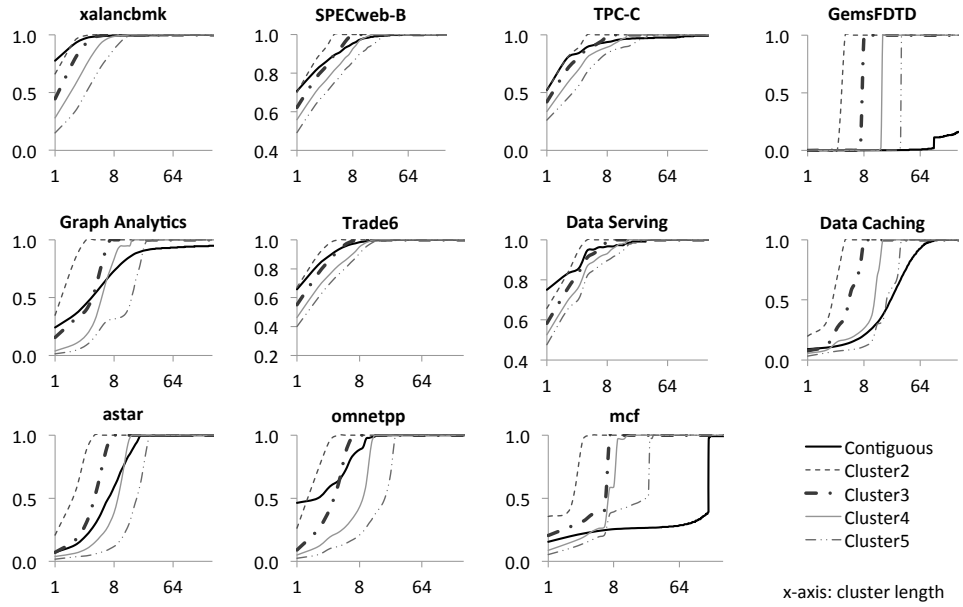


Figure 3.4: Cumulative distribution functions comparing the opportunity of CoLT-style contiguous spatial locality versus the clustered spatial locality that we target. In general, clustered spatial locality covers a bigger portion of the page table than contiguous spatial locality.

cases in which CoLT-styled sequentially allocated translations provide decent coalescing opportunities (e.g., GemsFDTD, mcf), and others in which sequentially consecutive translations are less prevalent (e.g., xalancbmk, SPECweb-B[2], Data Serving).

3.3.2 Clustered Spatial Locality

CoLT requires that consecutive virtual pages map to sequential physical pages. Requiring complete sequentiality for both VPNs and PPNs restricts coalescing opportunities. We instead consider the notion of clustered spatial locality: as long as nearby virtual pages map to nearby physical pages, we consider the corresponding PTEs coalescable. In Figure 3.4, the notation "Cluster$X$" indicates that translations from within an aligned cluster of 2^X virtual pages that map to an aligned cluster of 2^X physical pages potentially can be combined.

For example, the curves labeled Cluster3 limit the clustering or grouping of PTEs

[2] 'B' corresponds to the SPECweb banking workload.

to those that fall within the same aligned set of eight (i.e., 2^3) pages. With the CoLT-style contiguous curves, a value of (for example) 3 on the x-axis indicates that three consecutive PTE entries mapped to three consecutive physical pages. With the Cluster3 curves, the same value of three indicates that three virtual pages (from within a group of eight), map to pages within an aligned group of eight physical pages. The three VPNs need not be consecutive, and the corresponding three PPNs also do not need to be consecutive or even in increasing address order.

For each curve, the point at which the curve meets the y-axis (i.e., the y-intercept) indicates the percentage of PTEs that cannot be coalesced with any other PTEs. For most benchmarks, a modest clustering scope of Cluster2 or Cluster3 can uncover more opportunities for PTE coalescing than when using the more constrained CoLT-like contiguous criteria. For example, in the benchmark xalancbmk, a CoLT-like approach leaves 77% of translations uncoalesced, whereas when considering groups of four (Cluster2) or eight (Cluster3) pages without the sequentiality constraint, the percentage of uncoalescable PTEs drops to 66% and 45%, respectively. Furthermore, the exact curves are heavily dependent on benchmark behavior. For example, Data Caching sees particularly large amounts of contiguous spatial locality (and even clustered spatial locality) because it uses memcached, which in turn allocates large data structures using the slab allocator, which targets contiguous memory allocation.

In almost all cases, relaxed clustering allows significantly more coalescing opportunities than a CoLT-based approach. As the clustering scope increases (i.e., larger X for Cluster$X$), the opportunities for coalescing increase as well (curves move further down and to the right), but in many cases Cluster3 or Cluster4 are sufficient for capturing a significant portion of the opportunity. Mcf is a case when the CoLT-styled approach appears to provide significantly more coalescing opportunity; this arises because the Cluster$X$ criteria limits the coalescing scope to at most 2^X pages, whereas if CoLT gets lucky and there exists a very long run of adjacent translations, CoLT can coalesce all of these.

3.3.3 Impact of Memory System Fragmentation

A potential criticism and limitation of any TLB-coalescing approach is that after a system has been up and running for a long time, the virtual memory system may become fragmented. A highly-fragmented memory system would make it unlikely that nearby virtual pages get mapped to nearby physical pages (let alone to completely sequential physical pages). To quantify the impact of OS memory allocation fragmentation, we ran a subset of the benchmarks on a real server[3] and extracted live snapshots of the applications' page tables. This server is a highly-utilized machine mostly dedicated to running simulations, and the machine had an uptime of about 1.5 months at the time of these experiments. We found that while the exact amounts of coalescing opportunity are not the same as our trace-based analysis, they follow the same trends. We found that despite 1.5 month's worth of fragmentation, clustered spatial locality is more prevalent than contiguous spatial locality *in every single case*. For astar, omnetpp, and mcf, we found that eight clustered entries cover close to the full page table, whereas more than 64 such entries are required if only contiguous spatial locality is leveraged.

3.4 The Multi-granular TLB

The overall multi-granular TLB consists of a *clustered TLB* that can efficiently store multiple translations for PTEs with clustered spatial locality, a *conventional TLB* for singleton translations without spatial locality, *coalescing logic* for detecting clustered spatial locality and populating entries of the clustered TLB, and logic for performing look-ups, evictions, and other standard TLB operations.

3.4.1 Clustered TLB

Structure: The basic clustered TLB is a set-associative structure, much like a conventional TLB, but each TLB entry is designed to store multiple clustered page table translations. Figure 3.5(a) shows a clustered TLB entry. In this example, we assume a clustering reach of eight PTEs (i.e., Cluster3, or C3 for short). The eight VPNs all

[3] 32-thread x86 multiprocessor with 64GB memory, running 64-bit Ubuntu OS v12.04.

have identical values except for the lowest-three bits. The VPN's upper bits (i.e., the VPN's bits excluding the lowest three) are called the base VPN. Likewise, any of the coalescable PPNs are identical apart from their respective lowest-three bits, and the common prefix of the PPN is similarly called the base PPN. The key is that because all of the coalescable translations have identical base VPNs and base PPNs, each of these base values needs to be stored only once per clustered-TLB entry. Only the low-order bits of the PPN need to be tracked individually.

The C3 TLB entry potentially can track up to eight PTEs. For each of the individual potential translations, there is one *sub-entry*. Figure 3.5(a) also shows these eight sub-entries, with a detail of one such sub-entry's contents. Each sub-entry contains a valid bit, a dirty bit, a *referenced bit* (used in replacement policies described later), and the low-order bits of the PPN (e.g., the lowest-three bits in the case of Cluster3).

Look-up: To perform a look-up on the clustered TLB, we start with the VPN. Instead of using the entire VPN to generate a set index, we use only the base VPN (e.g., the lowest-three bits are omitted), as shown in Figure 3.5(b). If the requested base VPN matches the base VPN stored in the indexed clustered TLB entry[4], then we have a *cluster hit*, but this does not necessarily imply that the requested VPN is tracked by the clustered TLB. Next, we take the low-order bits of the VPN to select one of the eight sub-entries. If the selected sub-entry's valid bit is set, then this indicates an actual hit. The translated PPN then simply is reconstructed by concatenating the base PPN with the low-order PPN bits stored in the selected sub-entry. The sub-entry's referenced bit is set, and if the request corresponded to a write operation, then the sub-entry's dirty bit also is set. If we do not have a cluster hit, or if the selected sub-entry is not valid, then in either case this results in a cluster TLB miss.

Fill: On a clustered TLB miss, the TLB look-up is forwarded to the hardware page-table walker (PTW). In x86-64 processors, the PTW traverses the four-level page table and returns a cache line containing the requested PTE, as shown in Figure 3.6(a).

[4]For simplicity, only a direct-mapped clustered TLB is shown in the figure, but extension to a set-associative organization parallels that for a conventional TLB or cache.

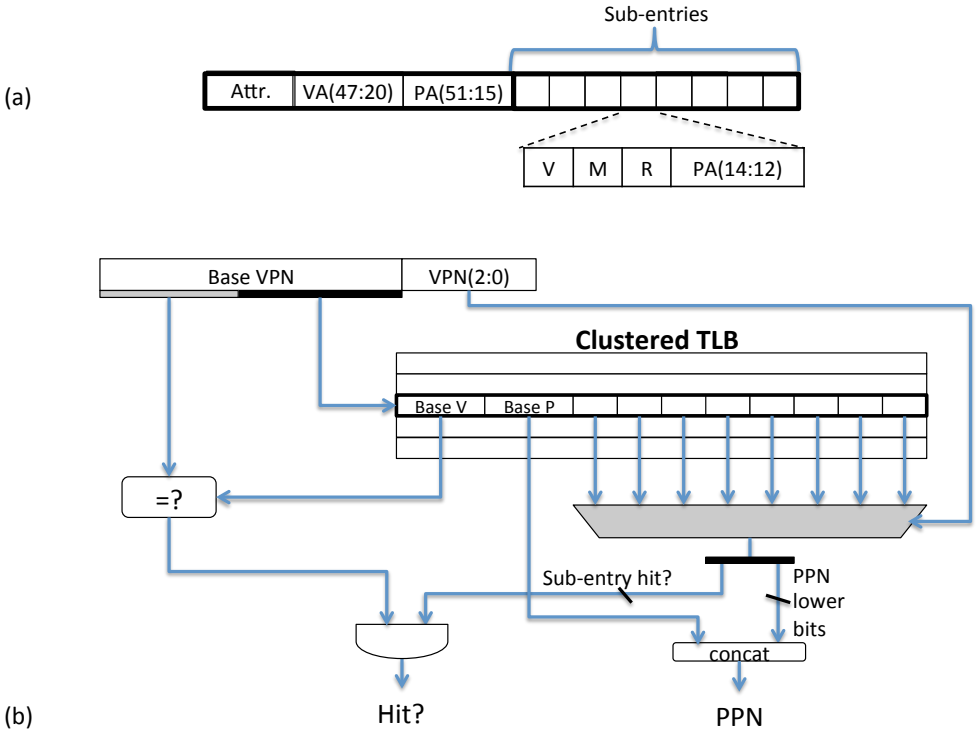


Figure 3.5: (a) Clustered TLB entry format, (b) Look-up operation

In x86-64, the PTE entry size is 8 bytes, which means that a single 64-byte cache line contains a total of eight PTEs (i.e., the requested PTE plus seven others). The clustered TLB's coalescing logic examines the seven non-requested PTEs and checks to see which of these can be coalesced (shown shaded in Figure 3.6(b)) with the originally requested PTE (i.e., it detects how many PTEs exhibit clustered spatial locality). For the original PTE and each coalescable PTE, the corresponding sub-entry will have the valid bit set, the referenced bit cleared, and the low-order PPN bits stored. All other sub-entries have their valid bits cleared. The common base VPN and base PPN are stored in the overall clustered-TLB entry. All of this logic is off the critical path because the originally requested PTE can be returned as soon as the PTW's page table look-up has completed.

Clustered TLB Eviction: For a set-associative clustered TLB, a clustered TLB entry first must be evicted prior to installing a new set of clustered PTEs. Each clustered

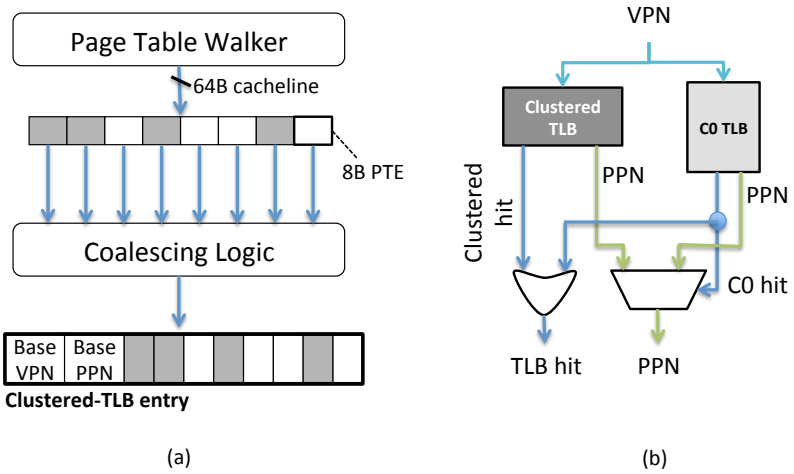


Figure 3.6: (a) Coalescing is performed on TLB fill, (b) Multi-granular TLB consists of a clustered TLB and a small conventional TLB.

TLB entry may contain a different number of valid translations; simply relying on conventional replacement policies such as LRU fails to account for situations when the LRU entry contains many valid translations and other more recently used entries may contain only a few. It is not immediately clear how to trade optimizing for recency against the retention of a larger number of translations.

The sheer number of *valid* translations in a clustered TLB entry might not reflect the utility of those translations. The coalescing logic may prefetch up to seven additional translations (assuming Cluster3), but it is possible that none of these other translations are needed. We propose a simple replacement algorithm that incorporates the referenced bits from each of the sub-entries to estimate the overall *usefulness* of the clustered TLB entry. Usefulness is the number of valid sub-entries with their referenced bits set. We also define a *recency* value, which is the clustered TLB entry's position in the LRU recency stack (lower value = less recently used). Then for each clustered TLB entry in a set, we compute a retention priority:

$$priority = (\alpha * recency) + (\beta * usefulness)$$

The clustered TLB entry with the lowest priority is selected as the victim. This provides a balance between the recency of the clustered TLB entry, and the number of *useful* translations stored by the entry. We found that setting α and β to 1 and 2, respectively,

provided good performance while maintaining very simple hardware (e.g., for a four-way set-associative clustered TLB, the recency value is only two bits wide, the usefulness value is four bits wide for Cluster3, and multiplication by 1 and 2 are trivial.

To avoid the pathological situation when a clustered TLB entry that has not been used recently, but still is kept around due to a large number of sub-entries that were useful (i.e., referenced) a long time ago, we periodically decay the referenced bits. We found that exactly how the referenced bits are cleared is not very important; we tried periodic and pseudo-random approaches across a very large range of decay intervals and found that overall performance is largely insensitive if *some* decay occurs every now and then.

3.4.2 Multi-granular TLB Organization and Operation

The clustered TLB provides efficient storage of multiple PTEs by *not* having to redundantly store multiple copies of the base VPN and base PPN for translations that exhibit clustered spatial locality. However, the locality characterization results from Section 4.3 showed that there remains a non-trivial percentage of PTEs that cannot be coalesced with other PTEs. Storing such singleton translations in a clustered-TLB entry would be wasteful because only a single sub-entry would be utilized. Figure 3.6(b) shows the high-level organization of the multi-granular TLB (MG-TLB). The MG-TLB consists of a clustered TLB paired with a conventionally-organized (i.e., not clustered) TLB. For shorthand, we label the conventional TLB "C0".[5] The conventional TLB is primarily utilized to cache singleton translations that cannot be clustered with other PTEs.

Look-up: To perform a look-up, both structures are searched in parallel. A hit in either indicates a TLB hit, and the translation is provided by the hitting structure. A miss in both structures results in an overall TLB miss, and the request is sent to the PTW to retrieve the translation from the page table.

[5] A conventional TLB can be viewed as a degenerate case of the clustered TLB with a clustering scope of zero (i.e., Cluster0).

Fill: In the MG-TLB, when the PTW returns the cache line with the requested PTE (along with the seven other neighboring PTEs), the entire set of eight PTEs is delivered to the clustered TLB's coalescing logic. At the end of the coalescing process, the MG-TLB first checks the coalescing degree (i.e., how many PTEs were successfully coalesced). If the coalescing degree is greater than a threshold value θ, then the entire set of coalesced translations is installed into the clustered TLB. Otherwise, the single originally-requested translation is installed into the conventional C0 TLB. The idea is that the clustered-TLB entries provide greater encoding density (i.e., valid translation per unit area) when the coalescing degree is high. If it is more efficient to store a few (less than θ) translations in C0, then that should be done instead.

Clustered TLB Eviction: When an entry is evicted from the clustered TLB, it is possible that it contains one or more valid translations. One option is simply to drop them all; if any are needed, a subsequent TLB miss will cause them to be re-fetched by the PTW. This could cause an increase in TLB misses. Another approach is to take them all and place each translation in individual entries of the conventional C0 TLB. This approach is unfortunately space-inefficient (which is why we clustered them in the first place). We instead take a middle-of-the-road approach in which only those sub-entries that have their referenced bits set are "saved" and installed into the C0 TLB. The remaining translations are dropped. Apart from preserving useful translations, this is an important optimization because it provides a way to convert a clustered-TLB entry with a high degree of coalescing but low actual usefulness into more efficiently stored conventional TLB entries (i.e., clustered spatial locality does not buy you anything if the coalesced entries are never used, and this policy allows such clustered TLB entries to be "de-coalesced").

3.4.3 Frequent Value Locality in the Address Bits

Our baseline clustered TLB design exploits the fact that the upper bits of the VPNs and PPNs of nearby PTE entries often contain the same values. We observed that this spatial locality in the addresses' bit patterns also occurs on a global scale.

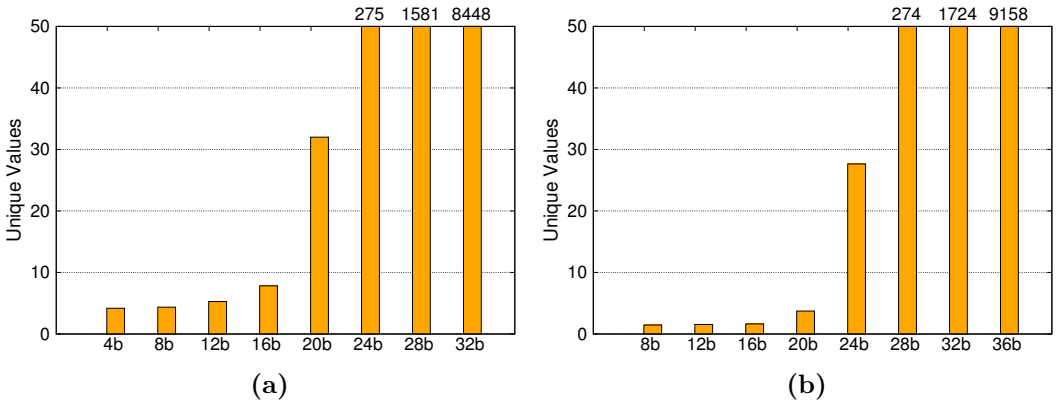


Figure 3.7: The number of unique values when only considering the x upper-most bits for the VPN (a) and PPN (b), as x is varied. The upper 16 VPN bits and 20 PPN bits change rarely in our experiments.

Consider the layout of typical virtual address spaces. A program's virtual memory space usually is partitioned into a few, large logical regions corresponding to the program's text, heap, stack, etc. These usually are contiguous in the virtual address space, which creates a few frequently used memory regions. When considering only the few most-significant bits of virtual addresses of all valid PTEs, we typically find only a few unique values.

Figure 3.7a quantifies this entropy by showing the number of unique values (y-axis) used by the most significant bits from the VPN (x-axis) on average across our benchmarks. For example, when considering the 12 most-significant bits of the VPN (we use 48-bit virtual addresses), on average we only observe five unique values. This is similar to past work in *frequent value locality* (FVL) that showed that memory locations often store values drawn from a small set of common values (e.g., zero) [76, 93]. In this case, we effectively observe that similar FVL exists in the upper bits of the VPNs.

In a similar fashion, Figure 3.7b shows that the most significant PPN bits tend only to use a few unique values. In particular, we find that the 20 most-significant bits of the PPN (we use 48-bit physical addresses) use only one of four unique values on average. This is due partially to the tendency of OSs to cluster physical pages so that transfers to disk make good use of high disk bandwidth.

We leverage that the most-significant VPN and PPN bits typically are drawn from

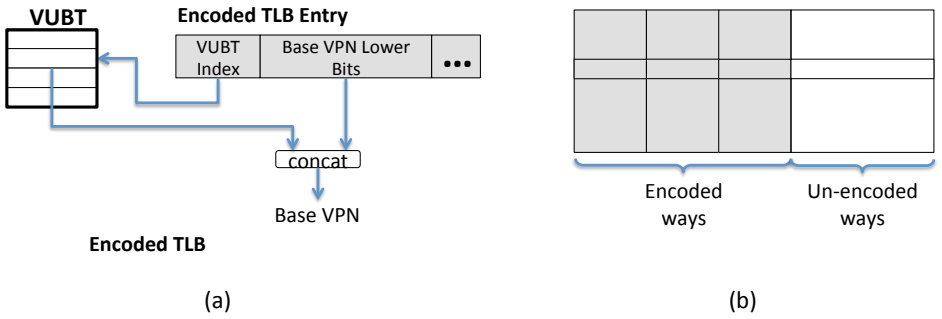


Figure 3.8: (a) Hardware organization for the Virtual Upper Bits Table (VUBT) and an encoded TLB entry, and (b) a four-way TLB with three encoded ways and one un-encoded way.

a limited set of unique values to optimize the MG-TLB further. This also can be used for the baseline TLB and CoLT.

FVL-Support for TLBs: We partition the TLB's base VPN field into the upper bits that tend to come only from a small set of unique values, and the lower bits that exhibit greater value diversity. Figure 3.8(a) shows an auxiliary structure called the *virtual upper bits table* (VUBT). This stores the commonly-occurring upper bits of the base VPNs. The TLB entry now is modified such that the VPN's upper bits are removed and replaced with an index into the VUBT. To reconstruct the entry's VPN, the VUBT index selects one of the VUBT entries that provides the upper bits of the VPN. The remaining bits of the VPN come from the TLB entry itself. A similar *physical upper bits table* (PUBT) encodes the upper bits of the PPNs. This can be applied to the clustered TLB as well as the conventional TLB. We call such an entry an *encoded* TLB entry.

The VUBT (or PUBT) is limited in size, and so if a VPN's (or PPN's) upper bits do not match any of the entries of the VUBT (PUBT), then the translation cannot be stored in an encoded TLB entry. To support situations when the number of unique VPN upper-bit values exceeds the capacity of the VUBT, we use a hybrid TLB structure in which some ways support the encoded scheme, and other ways use a conventional VPN format. Figure 3.8(b) shows an example four-way TLB in which the first three ways use encoded TLB entries, and the last way uses un-encoded entries.

Look-up: Look-up proceeds as with a conventional TLB in which the VPN (or base VPN) is used to select a set. For each encoded way, the VUBT index first is used to look up the VPN upper bits from the VUBT. This is concatenated with the stored lower bits to form the overall VPN (or, more accurately, the VPN tag). For the un-encoded ways, the entire VPN (tag) can be read directly from the TLB entries. At this point, each way now has a fully decoded VPN tag, and this can be compared to the requested VPN to determine if there is a TLB hit.

If there is a hit in an encoded entry, the stored PUBT index is used to select the upper PPN bits from the PUBT, which are then concatenated with the lower PPN bits stored in the encoded TLB entry. A hit in an un-encoded way simply uses the PPN already stored in that TLB entry.

Fill: On a TLB miss, the VUBT is searched to see if any existing entries match the upper bits of the VPN (for the translation we are installing into the TLB). At the same time, a similar search is performed on the PUBT for the upper bits of the PPN. If there are matches in *both* the VUBT and the PUBT, then the translation can be installed in an encoded TLB entry. If the upper bits cannot be found in one of the VUBT entries, then a new VUBT entry is allocated for this new upper-bit value. The translation is installed into an encoded TLB entry (assuming the PPN had a match in the PUBT) and the encoded entry stores the index of this newly allocated PUBT entry. A symmetric operation is performed if the PPN's upper bits do not match any existing PUBT entry. If a VUBT or PUBT entry cannot be allocated (i.e., the VUBT or PUBT is full), then the translation is installed into a un-encoded TLB entry.

VUBT and PUBT Management: When an application (process) is first context-switched onto a core, a new page table base pointer (i.e., CR3) is loaded and the TLB is flushed. At the same time, we also flush the VUBT and PUBT. As previously un-encountered VPN and PPN upper-bit values are encountered, they will be allocated new entries in the VUBT and PUBT, respectively. The entries are allocated in order; thus, instead of per-entry valid bits, a single allocation counter per table is needed.

Eventually, one or both of these may fill up, at which point any new VPN or PPN

upper-bit values will cause the corresponding translations to be restricted to the un-encoded entries in the TLB. Our characterization (Figure **??**) showed that typically there are only a few unique VPN and PPN upper-bit values, and so very small VUBT and PUBT sizes are needed in the vast majority of cases. Based on the characterization results, we set the VUBT size at eight entries and the PUBT size at four. A small VUBT table is desirable because to perform the encoded TLB look-up, each encoded way needs to perform a look-up on the VUBT, and therefore the VUBT needs to be multi-ported (each look-up is a RAM look-up). Similarly, on a fill operation, we need to check if the current VPN upper-bits are already present in *any* of the VUBT entries, which requires a single CAM port. Keeping the VUBT small makes the extra ports not very expensive. The PUBT is slightly simpler because only the hitting way needs to perform a look-up, and so it can be limited to a single RAM port and a single CAM port.

Even if the VUBT or PUBT fills up, translations with the upper-bit values not in these tables will continue to be cached in the TLB's un-encoded ways. The monotonic, write-only allocation of the VUBT/PUBT may seem like a problem, but these will be flushed on every context switch. If a single program runs for a very long time without ever being context-switched out by the OS, it would be trivial to have the processor periodically (but fairly infrequently) flush the TLB and VUBT/PUBT. The performance impact is minimal if this flush interval is sufficiently long. For a simultaneous multi-threaded (SMT) processor, we could have one set of UBTs per hardware thread. This avoids flushing all UBTs when only a single thread is context-switched. Given the small size of the UBTs, the space overhead is minimal (typical SMTs are only two-threaded).

3.4.4 Hardware Cost

3.4.4.1 Basic Multi-granular TLB Hardware Cost

Table 3.1 compares area cost and reach for conventional TLB, CoLT-SA (the set-associative version of the CoLT TLB proposed by Pham et al. [70]), and the MG-TLB. *Conventional TLB:* 512 entries, four ways. Each entry has 29 bits for the tag, 40 bits for the PPN, and 5 bits for the attribute. In total, we have 75 bits per entry, which

Table 3.1: Comparison of Hardware Cost

	Baseline L2TLB	CoLT-SA	Cluster-C3	Cluster-C0
Entries	512	512	128	320
Assoc.	4	4	4	4
Max Reach	512	2,048	1,024	320
Min Reach	512	512	128	320
Attr. Bits	5	5	5	5
Tag Bits	29	27	28	30
Data Bits	40	48	85	40
Entry Bits	74	80	118	75
Total Bits	37,888	40,960	15,104	24,000

adds up to 37,888 bits (4.625KB). The other designs are configured to target a similar bit-storage budget.

CoLT-SA: 512 entries, four ways. Each CoLT entry has only 27 bits for the tag because we left-shift the VPN by 2 bits to compute the set index; 40 bits for the base PPN; 8 bits for 4 sub-entries, each sub-entry in the contiguous range has 1 valid bit and 1 dirty bit, and 5 bits for the attribute. As a result, each CoLT entry has 80 bits, which gives us 40,960 bits total, or 8% area overhead compared to the baseline.

Multi-granular TLB: We allocate about one-third of the storage budget to C3 and two-thirds of the budget to C0. This results in 128 C3 entries, and 320 C0 entries[6]. Each C3 entry has 5 bits for the attribute; 28 bits for the tag because we ignore the bottom 3 bits of the VPN; 37 bits for the base PPN because we also ignore the bottom 3 bits of the PPN; and, eight sub-entries, with each sub-entry having 6 bits (valid, modified, referenced, and 3 bottom bits of the corresponding PPN). Hence, 128 C3 entries requires 15,104 bits. Each C0 entry has 5 bits for the attribute, 30 bits for the tag, and 40 bits for the PPN. Therefore, 320 C0 entries require 24,000 bits. This MG-TLB configuration requires 39,104 bits which is close to the baseline (3%) and less than CoLT.

Given these configurations, the conventional TLB always has a reach of 512 pages.

[6]In a real implementation, the number of C0 entries (or at least the number of sets) would be a power-of-two. For the purposes of maintaining similar storage budgets across each type of TLB for fair comparisons, we used a non-power-of-two size.

Table 3.2: Enhanced MG-TLB Hardware Cost

	C3's Full Len Way	C3's Encoded Way	C0's Full Len Way	C0's Encoded Way
Entries	38	114	109	327
Max Reach	304	912	109	327
Min Reach	38	114	109	327
Attr. Bits	5	5	5	5
Tag Bits	29	12	31	14
Data Bits	85	69	40	24
VEncode Bits	3	3	3	3
PEncode Bits	2	2	2	2
Entry Bits	119	91	76	48
VUBT Bits (Shared)	132	132	132	132
PUBT Bits (Shared)	67	67	67	67
Total Bits	4,522	10,374	8,284	15,696

CoLT-SA can have a coverage of up to 2,048 pages (i.e., if each of the 512 entries fully coalesces four PTEs), while the MG-TLB has maximum reach of 1,024 in the C3 table (i.e., if each of the 128 clustered-TLB entries is fully populated with eight translations) plus the 320 entries in the C0 TLB for a total of 1,344 possible translations. In the worst case, CoLT-SA has reach of 512 pages, while the MG-TLB has a reach of 448. At first glance, CoLT-SA may appear to be better in terms of reach than the MG-TLB, but this is true only if CoLT-SA can find enough contiguous spatial locality. We will show that the MG-TLB's effective reach is superior to CoLT-SA's because, in practice, clustered spatial locality is easier to find than strict contiguous locality.

3.4.4.2 Enhanced Multi-granular TLB Hardware Cost

Table 3.2 shows the configuration of the MG-TLB when we exploit the FVL in the upper bits of VPNs and PPNs. We use a similar relative area allocation between C3 and C0 as before; however, for each encoded TLB entry, we need to keep additional

bits for the VUBT and PUBT indexes. We assume a VUBT with eight entries, and a PUBT with four entries; therefore, the respective indexes are 3 and 2 bits each. In addition, a small part of the area budget is used to implement the VUBT and PUBT (along with one small allocation counter for each). Each VUBT or PUBT entry is 16 bits and the VUBT and PUBT counters are 4 and 3 bits, respectively, so in total we need 132 bits for the VUBT and 67 bits for the PUBT. Overall, the total area cost is 39,075 bits, or **3%** of area overhead compared to the baseline, which also is much less than CoLT.

Despite those additional bits, by replacing the 16 upper bits of the VPN or the PPN with a 3-bit VUBT or 2-bit PUBT index, respectively, we can save quite a bit of space, which allows us to add more entries to both the C3 and C0 TLBs. While maintaining approximately the same bit-budget as the un-encoded MG-TLB, C3 and C0 TLBs that each use three encoded ways allow us to have 152 C3 entries and 436 C0 entries. This increases the maximum reach of the MG-TLB to 1,652 (from 1,344 in the basic design) and the minimum reach from 448 to 588. We will show that this design performs the best out of all of our evaluated configurations.

3.5 Experimental Methodology

This section describes the infrastructure[7] used to evaluate our multi-granular TLB and two comparison points: a baseline conventional TLB and the recently proposed CoLT design.

3.5.1 Workloads

Table 3.3 shows the workloads evaluated in our study. We consider a wide range of applications, from scientific workloads to server and cloud workloads, and select benchmarks with non-negligible TLB miss overheads. We also evaluated eight benchmarks with low TLB sensitivity (from SPECcpu 2006, SPECjbb2005, web browsing, and gaming; results for these benchmarks are omitted for space), and their results are consistent

[7] I was using this simulation infrastructure during my internship at AMD Research, from 1/2013 to 8/2013.

Table 3.3: Summary of benchmarks used in our studies

Benchmarks	Suite	Page Walk Overhead
xalancbmk	SPEC CPU2006	9.4%
SPECweb-B	SPECweb2005	9.5%
TPC-C	TPC	8.6%
GemsFDTD	SPEC CPU2006	9.2%
Graph Analytics	CloudSuite	17.7%
Trade6	IBM WebSphere	11.1%
Data Serving	CloudSuite	8.8%
Data Caching	CloudSuite	20.0%
astar	SPEC CPU2006	19.5%
omnetpp	SPEC CPU2006	26.4%
mcf	SPEC CPU2006	33.8%

with the observations we show in this work. We collect traces of at least 50 million instructions per benchmark using AMD's SimNow^{TM} [18] full-system simulator software. The traces capture issued user and system instruction and data references and record the virtual and physical page address pairs. We also correlate the traces against hardware performance counters to ensure that they capture a representative execution phase of the benchmarks.

3.5.2 Simulation Infrastructure

3.5.2.1 Functional Simulator

For fast design-space exploration of our multi-granular TLB in comparison to the baseline TLB and CoLT, we use a functional simulator that models a two-level TLB with 64-entry L1 instruction and data TLBs. We assume a baseline L2 TLB of 512 entries, similar to current products [70]. Because our multi-granular TLB targets the L2 level, we compare this against the benefits of CoLT on just the L2 TLB. All TLBs have four-way associativity.

3.5.2.2 Performance Evaluation

We use an in-house trace-driven timing simulator derived from the MacSim simulator [35], using a two-wide in-order core. It models three-level cache hierarchies, two-level TLBs, a hardware page-walk unit complete with a page-walk cache, and a detailed DRAM model. The TLBs have associated miss status holding registers (MSHRs) to

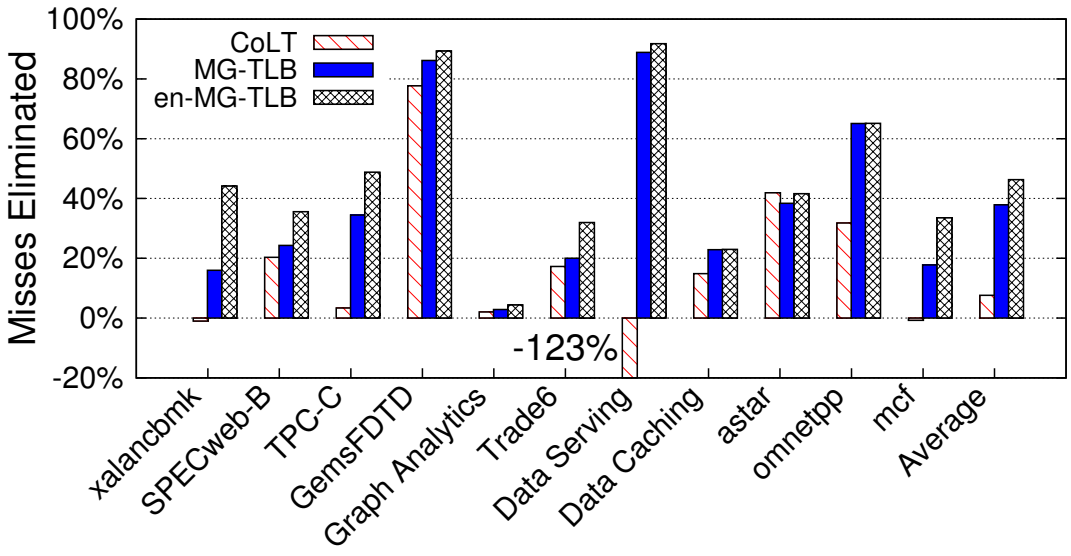


Figure 3.9: L2 TLB misses eliminated by the baseline multi-granular TLB (MG-TLB), enhanced MG-TLBs with structures to exploit redundant most significant VPN and PPN bits (en-MG-TLB) and CoLT. MG-TLB and en-MG-TLB comprehensively eliminate more misses than CoLT.

model pipelined accesses. We also calibrated the page-walk overheads of our timing simulator against hardware measurements on a real machine. As shown in Table 3.3, overheads for our benchmarks range from a problematic 9% for Data Serving to a severe 34% for mcf.

3.6 Multi-granular TLB Evaluations

Our multi-granular TLB (MG-TLB) enjoys a number of design options. We evaluate the performance implications of various options in this section.

3.6.1 Understanding Changes in Hit Rates

Figure 3.9 compares the percentage of L2 TLB misses eliminated (compared to a standard baseline four-way, 512-entry TLB) when using CoLT (512-entry); MG-TLBs without exploiting the frequent value locality in the VPN and PPN's upper bits (128-entry C3 TLBs with a 320-entry conventional C0 TLB); and the encoded MG-TLBs (en-MG-TLB) that leverage the upper-bit frequent value locality (152-entry C3 TLBs with 436-entry standard C0 TLB).

MG-TLBs eliminate more TLB misses than CoLT, averaging 38% miss eliminations (30% more than CoLT). FVL-based encoding (en-MG-TLB) only boosts this difference, eliminating on average 46% of the TLB misses. More specifically, we note the following three observations:

First, benchmarks that exhibit more clustered spatial locality than contiguous spatial locality (e.g., Data Serving, TPC-C) also eliminate more misses with MG-TLB than with CoLT. In fact, CoLT actually provides a negative result for Data Serving, mostly because the change in set-indexing scheme outweighs its ability to exploit contiguous spatial locality. Fortunately, exploiting clustered spatial locality overcomes this issue, eliminating the vast majority (more than 80%) of the TLB misses. Enhancements provide additional benefits.

Second, benchmarks like mcf or Data Caching, which show more contiguous spatial locality, *still benefit more from MG-TLB than CoLT*. We find that this occurs because changes to CoLT's set-indexing scheme undo the benefits of exploiting contiguity. Instead, our dual approach of leveraging clustered spatial contiguity and allowing a small conventional L2 TLB for singleton PTEs is more beneficial. As a result, MG-TLB eliminates 20% more of mcf's TLB misses than CoLT.

Third, en-MG-TLB improvements relative to MG-TLB are non-trivial for xalancbmk, TPC-C, Trade6, and mcf. In these benchmarks, the FVL in the upper bits makes even relatively small VUBT and PUBT tables highly effective.

3.6.2 Overall Performance Improvements

Figure 3.10 compares the performance improvement of en-MG-TLB with CoLT. Our approach outperforms CoLT in every case except astar. In some cases, the performance difference is significant (close to 18% for Data Serving, 12% for mcf, and 10% for omnetpp). On average, we outperform CoLT by about 5%, but we purposefully included benchmarks in which neither has much benefit (e.g., Graph Analytics) to demonstrate that the MG-TLB approach does not *hurt* performance when the spatial locality is sufficient, as well as benchmarks in which CoLT truly does well (e.g., astar, in which CoLT performs slightly better than MG-TLB). As TLB overheads continue to rise [16,

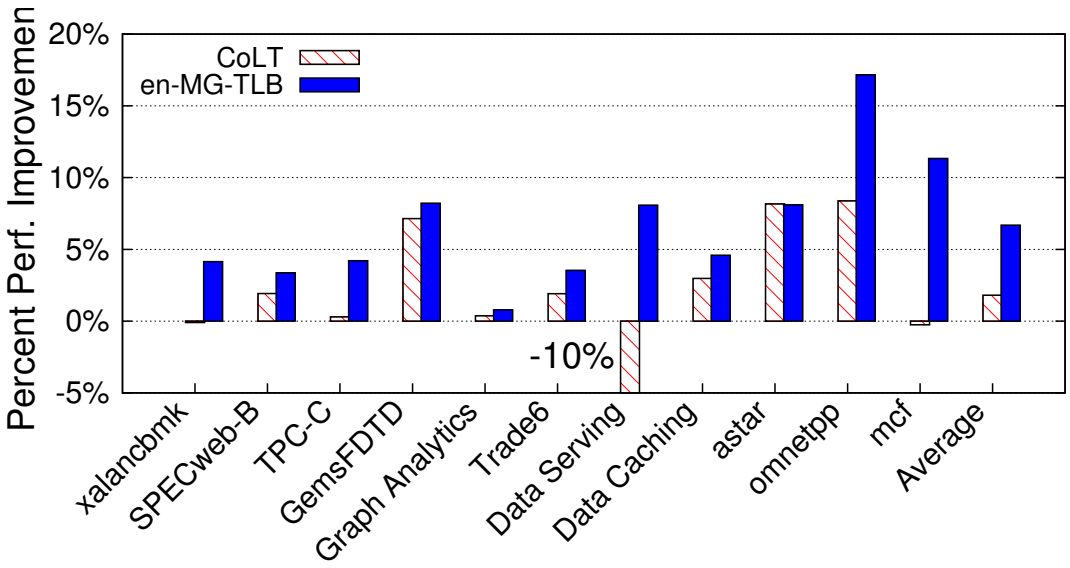


Figure 3.10: Performance improvements when using CoLT and en-MG-TLB. Our approach outperforms CoLT *in every single case*.

19], the expected benefit of TLB coalescing techniques such as MG-TLB would be expected to increase.

3.6.3 Prefetching versus Capacity Improvements

Our multi-granular TLB eliminates misses in two main ways. First, on a TLB miss, it speculates that PTEs near the one that is requested may be useful. This benefit is similar to prefetching because it is not known whether clustered PTEs will be useful in the future. However, unlike classical prefetching, which must evict an existing entry to make room for a new one, clustered TLBs use the same entry for the entire clustered packet. Second, because each clustered entry provides a higher reach, there is a capacity improvement relative to the standard approach, for the same total area.

Figure 3.11 teases apart the relative benefits of these two factors by plotting (1) TLB miss-elimination rates for MG-TLBs; (2) TLB miss elimination when the same clustered PTEs are prefetched into a standard 512-entry L2 TLB; and, (3) a lazy MG-TLB approach, in which only the desired PTE is inserted into the clustered TLB on demand, but which then integrates the other PTEs in the cluster if they are demanded in the future. Overall, this comparison informs us whether most of en-MG-TLB's benefits arise from prefetching effects or its superior reach.

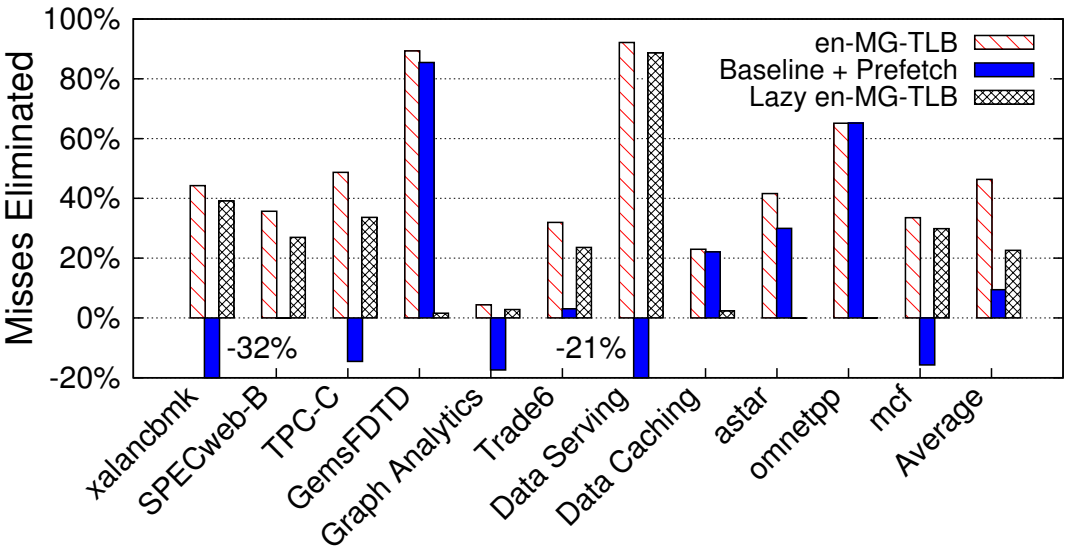


Figure 3.11: Separating the prefetch and capacity benefits of MG-TLBs.

Figure 3.11 shows that the relative benefits vary per benchmark. Some benchmarks (e.g., GemsFDTD, Data Caching, omnetpp) gain from prefetching. In fact, for some of these (e.g., omnetpp), the capacity benefit is almost negligible. The other benchmarks however, en-MG-TLB and the Lazy (no prefetch) version perform similarly, making clear that capacity is the main benefit. For some benchmarks, prefetching alone is negative (xalancbmk, Data Serving, etc.) because capacity improvement is the key to overall performance boosts.

3.7 Sensitivity Studies

MG-TLB has a number of parameters crucial to its overall performance. We investigate these parameters in this section.

Cluster Size: We have thus far assumed that the MG-TLB uses a C3 clustered TLB. Figure 3.12 shows how TLB miss-elimination rates change when C2 and C4 TLBs are used instead. Larger clustering potentially exploits more clustered spatial locality. At the same time, each entry's size increases, decreasing the total number of entries. Moreover, the selected index bits are further left-shifted, increasing the possibility of conflict misses when clustered spatial locality is insufficient.

Figure 3.12 shows that C3 tends to perform best on most benchmarks. In some

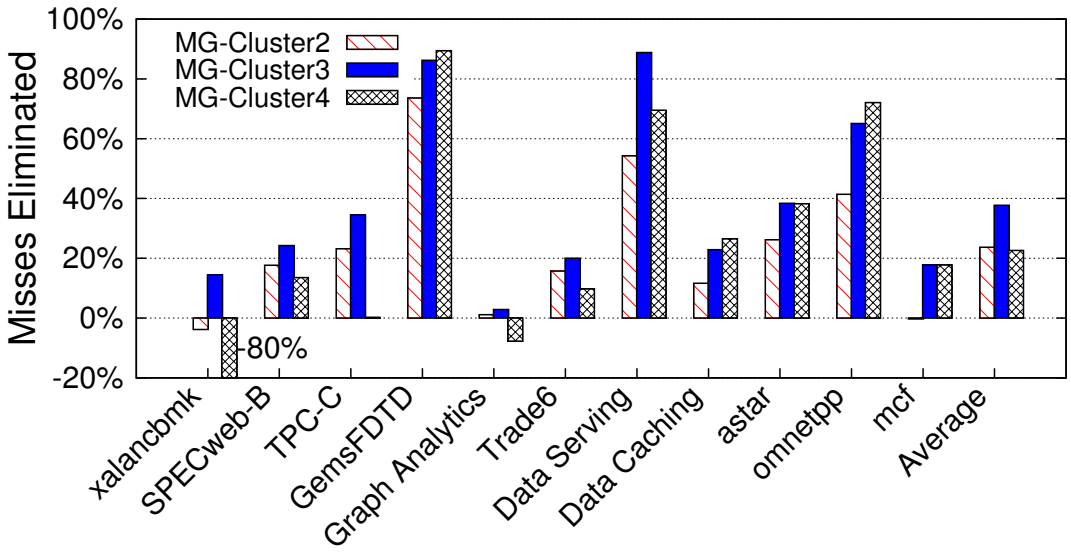


Figure 3.12: TLB miss-elimination rates assuming that the clustered TLB is C2, C3 (our default assumption so far), and C4

.

cases like data Caching, which is known to generate large clustered spatial locality due to slab allocator use, C4 outperforms C3. However, benchmarks like Data Serving and xalancbmk are degraded at C4.

Coalescing Thresholds: Our MG-TLB designs have assumed that at least $\theta = 2$ PTEs must be clustered for insertion into the clustered TLB. Figure 3.13 shows how this assumption affects miss rates by varying θ from 1 to 4 for en-MG-TLB with a C3 clustered TLB. We see that a value of 2 is generally the best (and is markedly better for some benchmarks like Data Serving and xalancbmk). Intuitively, this makes sense because a single C3 clustered TLB consumes less space than two conventional L2 TLB entries. Therefore, if we coalesce two PTEs and place them in a single C3 entry, we expend fewer bits in storing them compared to the small conventional L2 TLB for singleton PTEs. When θ goes beyond 2, these two PTEs are stored in two separate C0 TLB entries, wasting space.

Sizing MG-TLB Components: We now consider how the relative sizes of the MG-TLB's conventional small L2 TLB and clustered L2 TLB affect TLB miss rates. Our default scheme devotes about one-third of MG-TLB area for the clustered TLB and two-thirds for the conventional L2 TLB. Figure 3.14 shows how TLB miss eliminations vary when

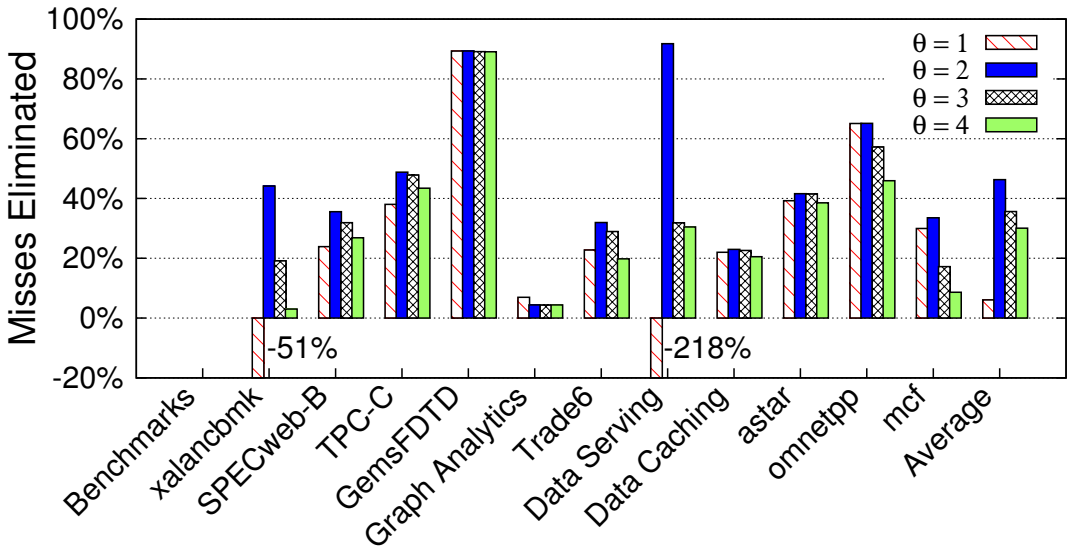


Figure 3.13: TLB miss-elimination rates for en-MG-TLB as the cluster threshold is changed for insertion into the clustered TLB.

these values are changed. The x:y ratio tells us what portion of the MG-TLB area goes to the conventional L2 TLB (x) and the clustered TLB (y). Generally, we find that TLB misses are best for our default configuration (2:1), though other configurations see similar gains. However, MG-TLB effectiveness perceptibly diminishes when the conventional L2 TLB becomes much smaller in comparison (e.g., 1:4), indicating that we must adequately cache the singleton PTEs that do not experience clustered spatial locality.

Sensitivity to VUBT and PUBT Size: Our default en-MG-TLB uses 8-entry PUBTs and 4-entry VUBTs. We have varied these sizes to study their impact on TLB miss rates. In general, PUBTs rarely require additional entries, whereas VUBTs require the use of the dedicated full-length way in rare instances. Overall, even 128-entry VUBTs and PUBTs provide negligible performance improvements to our approach.

MG-TLB Effectiveness for Different Sizes: Our evaluations compare MG-TLB to a baseline 512-entry L2 TLB. Therefore, all our designs are sized to meet this total area. However, we also have studied cases in which we have half and double the total area to play with (i.e., our baseline L2 TLB becomes 256 or 1,024 entries). We find that MG-TLB (and its encoded counterparts) consistently outperforms both the baseline L2

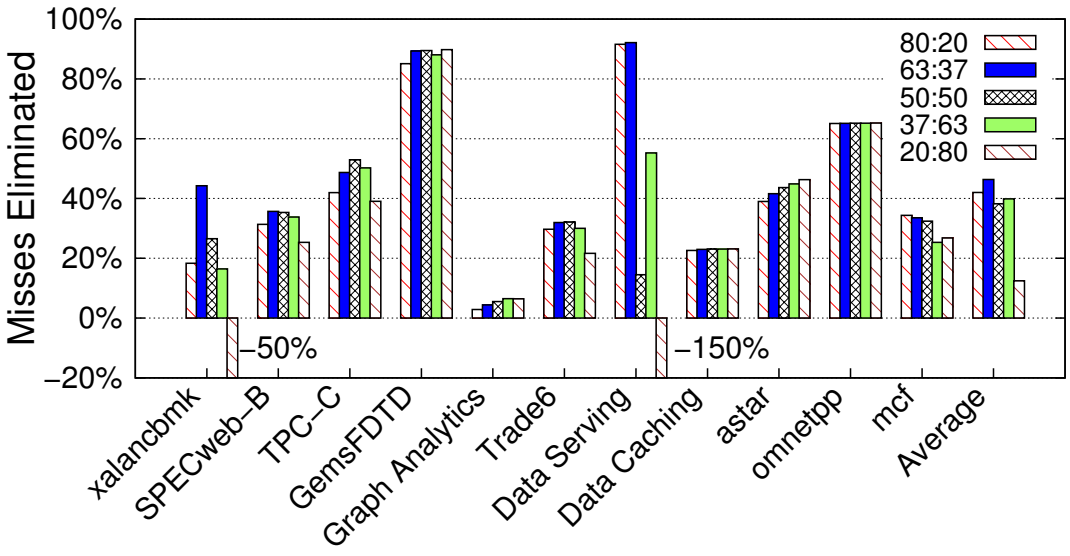


Figure 3.14: TLB miss eliminations for different relative sizes of the small singleton PTE's TLB and the clustered TLB in MG-TLB. The legend shows the ratio of the MG-TLB area for the small conventional TLB to the area for the clustered TLB.

.

TLB and CoLT for these sizes, and that its performance benefits increase when more area is available for some benchmarks (e.g., mcf, Data Serving). This bodes well for future designs which will likely have more resources available for address translation.

3.8 Summary

One of the main contributions of this thesis chapter is the observation that significant amounts of clustered spatial locality exists in applications. This form of spatial locality is present across a variety of system use cases and configurations (e.g., even in fragmented systems) and largely subsumes previously-observed contiguous spatial locality. In response, we propose a multi-granular TLB that identifies PTEs in which groups of nearby virtual pages are mapped to groups of nearby physical pages. By coupling a clustered TLB for these types of PTEs with a small conventional L2 TLB, we consistently outperform past work on coalesced TLBs despite using modest hardware and requiring no dedicated software support.

Our best-performing design point eliminates 46% of L2 TLB misses, resulting in a 7% CPU cycle reduction for a wide range of applications. Our proposed TLB organization substantially increases the effective TLB reach with only modest hardware changes

while requiring no OS support, providing a promising solution for emerging big data applications.

Chapter 4

Supporting Large, Yet Agile Pages in Virtualized Systems

4.1 Introduction

With cloud computing, virtualization technologies (e.g., KVM, Xen, ESX, Docker, HyperV and others) are aggressively employed by companies like Amazon or Rack-space to consolidate diverse workloads (encapsulated in virtual machines or VMs) to ensure high utilization of physical systems while achieving good performance.

The judicious use of large pages [64, 83, 86] is particularly critical to the performance of cloud environments. Large pages can boost address translation performance in hypervisor-based virtualization and containers [19]. Specifically, in hypervisor-based virtualization, the hypervisor and guests maintain separate page tables, and therefore require high-latency two-dimensional page table walks on Translation Lookaside Buffer (TLB) misses. Past work has shown that this is often the primary contributor to the performance difference between virtualized and bare-metal performance [26, 34]. Large pages (e.g., 2MB pages instead of baseline 4KB pages in x86-64) counter these overheads by dramatically reducing the number of page table entries (PTEs), increasing TLB hit rates and reducing miss latencies. Even containers (which do not require two-dimensional page table walks) are dependent on large pages to improve TLB reach, which is otherwise inadequate to address the hundreds of GB to several TB of memory present on the systems containers are often deployed in [34].

Unfortunately however, the coarser granularity of large pages can curtail lightweight and agile system memory management. For example, large pages can reduce consolidation in real-world cloud deployments where memory resources are over-committed

by precluding opportunities for page sharing [38, 90]. They can also interfere with a hypervisor's ability to perform lightweight guest memory usage monitoring, its ability to effectively allocate and place data in non-uniform memory access systems [36, 92], and can hamper operations like agile VM migration [88]. As a result, virtualization software often (though it doesn't have) chooses to *splinter* or break large pages into smaller baseline pages. While these decisions may be appropriate for overall performance as they improve consolidation ratios and memory management, they do present a lost opportunity in terms of reducing address translation overheads.

In this chapter, we propose hardware that bridges this fundamental conflict to reclaim the address translation performance opportunity lost by large page splintering. Specifically, we observe that the act of splintering a large page is usually performed to achieve finer-grained memory management rather than to fundamentally alter virtual or physical address spaces. Therefore, the vast majority of constituent small pages retain the original contiguity and alignment in both virtual and physical address spaces that allowed them to be merged into large pages in the first place. In response, we propose **Generalized Large-page Utilization Enhancements (GLUE)** to identify splintered large page-sized memory regions. GLUE augments standard TLBs to store information that identifies these contiguous, aligned, but splintered regions. GLUE then uses TLB speculation to identify the constituent translations. Small system physical pages are speculated by interpolating around the information stored about a single *speculative large-page* translation in the TLB. Speculations are verified by page table walks, now removed from the processor's critical path of execution, effectively converting the performance of correct speculations into TLB hits. GLUE accurate, software-transparent, readily-implementable, and allows large pages to be compatible, rather than at odds, with lightweight memory management. Specifically, our contributions are:

- We characterize the prevalence of page splintering in virtualized environments. We find that large pages are conflicted with lightweight memory management across a range of hypervisors (e.g., ESX, KVM) across architectures (e.g., ARM, x86-64) and container-based technologies.

- We propose interpolation-based TLB speculation to leverage splintered but well-aligned system physical page allocation to improve performance by an average of 14% across our workloads. This represents almost all the address translation overheads in the virtualized systems we study.
- We investigate design trade-offs and splintering characteristics to explain the benefits of GLUE. We show the robustness of GLUE, which improves performance in every single workload considered.

4.2 Background

Virtualization and TLB overheads: In hypervisor-based virtualized systems with two-dimensional page table support, guests maintain page tables mapping guest virtual pages (GVPs) to guest physical pages (GPPs), which are then converted by the hypervisor to system physical pages (SPPs) via a nested or extended page table [19]. TLBs cache frequently used GVP to SPP mappings; on TLB misses, the hardware page table walker performs a two-dimensional traversal of the page tables to identify the SPP. In x86-64 systems, because both guest and hypervisor use four-level radix-tree page tables, accessing each level of the guest's page table requires a corresponding traversal of the nested page table. Therefore, while native page table walks require four memory references, two-dimensional page table walks require twenty-four [19], significantly degrading system performance. Beyond hypervisors, container-based technologies also suffer from address translation overheads (even though they use standard one-dimensional page table walks), primarily because TLB reach is unable to match main memory capacities.

Our work focuses on hypervisor-based virtualization as it presents the greater challenge on address translation. However, we have also characterized page splintering on containers and present these results.

Large pages and address translation: To counter increased TLB overheads in virtual servers, virtualization vendors encourage using large pages aggressively [26]. OSes construct large pages by allocating a sufficient number of baseline contiguous virtual page frames to contiguous physical page frames, aligned at boundaries determined by

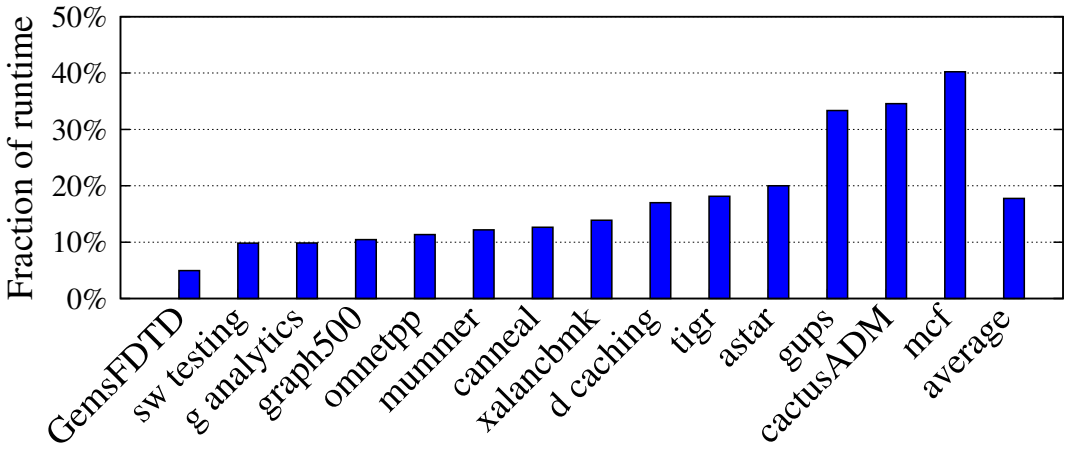


Figure 4.1: Percent of execution time for address translation, for applications on a Linux VM on VMware's ESX server, running on an x86-64 architecture. Overheads are 18% on average despite the fact that the OS uses both 4KB and 2MB pages.

the size of the large page [10]. For example, x86-64 2MB large pages require 512 contiguous 4KB baseline pages, aligned at 2MB boundaries. Large pages replace multiple baseline TLB entries with a single large page entry (increasing capacity), and reduce the number of levels in the page table (reducing miss penalty).

In hypervisor-based virtualization, these benefits are magnified as they are applied to two page tables. While a large page reduces the number of page table walk memory references from four to three in native cases, the reduction is from twenty-four to fifteen for virtual machines. However, because TLBs cache guest virtual to system physical translations directly, a "true" large page is one that is large in both the guest and the hypervisor page table.

4.3 Motivation and Our Approach

Real-system virtualization overheads: We begin by profiling address translation overheads on hypervisor-based virtualization technologies. Figure 4.1 quantifies address translation overheads (normalized to total runtime) when running a Ubuntu 12.04 server (3.8 kernel) as the guest operating system and VMware's ESX 5.5 as the hypervisor on an Intel Sandybridge architecture. Although omitted here for space reasons, we have also assessed these overheads running KVM on an x86-64 system, KVM on an ARM Cortex™A15 system, and we observed the same trends. We use SPECcpu®,

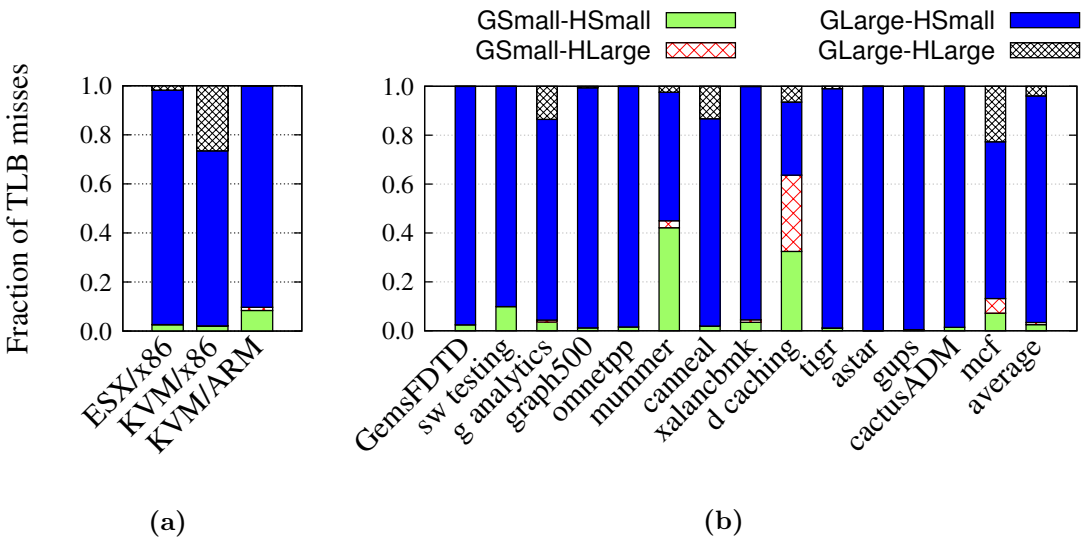


Figure 4.2: Fraction of TLB misses serviced from SPPs backed by a small page in guest and hypervisor (GSmall-HSmall), small in guest and large in hypervisor (GSmall-HLarge), large in guest and small in hypervisor (GLarge-HSmall), and large in both (GLarge-HLarge).

PARSEC [24], and CloudSuite [32] workloads and all measurements use on-chip performance counters.

The data show that two-dimensional page table walks degrade performance, consuming almost 20% of runtime on average. Overheads vary, with `graph analytics` from CloudSuite and `graph500` suffering 10% overheads while `mcf` and `cactusADM` suffer well over 30% overheads. Our results corroborate past work [26] that identified TLB overheads as a significant performance bottleneck in virtualized servers.

The prevalence of page splintering: Figure 4.2a quantifies the prevalence of splintering across different hypervisors and architectures, showing the generality of this problem. We profile splintering for ESX and KVM on x86-64 architectures, and KVM on ARM architectures. In all cases, we run four VMs, each with the workload shown. For each configuration's TLB misses, we plot the fraction eventually serviced from splintered large pages (large pages in the guest and small pages in the host), small pages in both dimensions, large pages in both dimensions, and small pages in the guest and large pages in the host (which is typically rare). Our results show that guest VMs construct and use large pages aggressively (on average, almost all TLB misses are to regions with large guest pages). However, the vast majority of these references are to guest

large pages that are splintered (GLarge-HSmall), regardless of the specific hypervisor or architecture used.

Figure 4.2b sheds light on the per-benchmark characteristics of page splintering. We find that guest OSes, when running workloads like `graph analytics`, `canneal`, and `mcf`, are able to more aggressively use large pages (usually because they allocate large data structures at once) but the hypervisor still chooses to splinter many pages for overall lightweight memory management. Going beyond hypervisor-based virtualization, we have also profiled page splintering in containers, using Linux containers with kernel same-page merging (KSM) [11] extensions to encourage high consolidation ratios. Similar to several real-world deployments [51] and we see that containers also suffer from large page splintering. This occurs because KSM shares pages among containers to consolidate as many workloads as possible with over 85% of TLB misses to splintered regions. The tension between large page address translation benefits and fine-grained memory management is regrettable because modern OSes work hard to create large pages.

Our high-level approach with GLUE: We illustrate GLUE's operation using hypervisor-based virtualization though it is also applicable to containers. We observe that page splintering usually splits large page sized regions into smaller pages for finer-grained management and monitoring, *without relocating most small pages.* Therefore, the contiguous memory alignment required to generate large pages likely remains; GVPs and SPPs are aligned within 2MB memory regions corresponding to the alignment they would have had in an unsplintered page. GLUE exploits these cases where GVPs and SPPs share their 9 least significant bits (for 4KB baseline pages and 2MB large pages). We have profiled page tables in many real-world deployments and in every case, we have found that over 82% (average of 97%) of the 4KB pages have GVPs and SPPs sharing their bottom 9 bits. These numbers foreshadow the potential for careful interpolation-based TLB speculation.

Figure 4.3 shows GLUE's operation. For illustrative purposes we assume that four contiguous PTEs make a large page, hence the guest page table can combine PTEs for

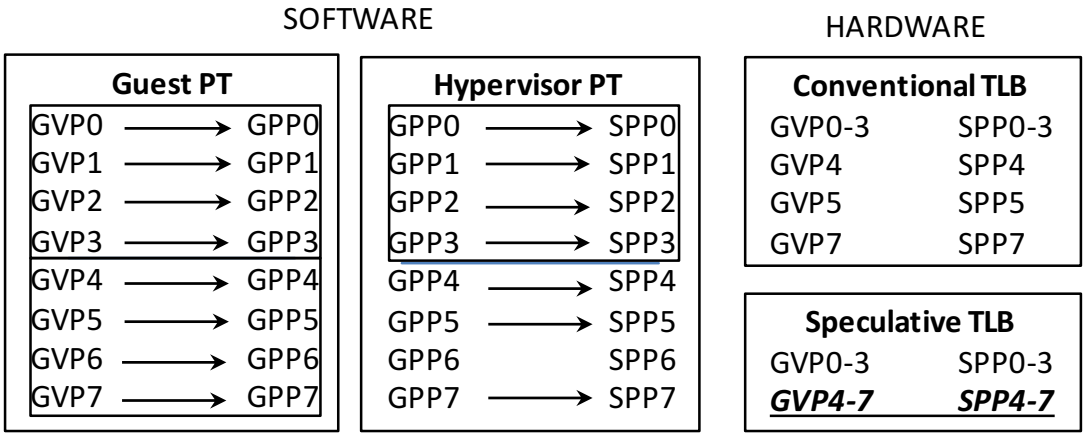


Figure 4.3: Guest large page GVP4-7 is splintered, but SPPs are conducive to interpolation. A speculative TLB maps the page table in two entries (using a speculative entry for GVP4-7) instead of four entries (like a conventional TLB).

GVP 0-3 into a single large page (the same for PTEs for GVP 4-7). The hypervisor does indeed back GVP 0-3 with its own large page (corresponding to the PTEs for GPP 0-3). Unfortunately, it splinters the guest large page for GVP 4-7 because GPP 6 (and SPP 6) are unallocated. A conventional TLB requires one large page entry and three baseline entries to fully cover this splintered large page. GLUE, on the other hand, observes that the SPPs corresponding to GVP 4-7 are still aligned in the way they would be had they actually been allocated a large page. Consequently, GLUE requires just two TLB entries to cover the page table, with one entry devoted to a speculative 2MB region (italicized and in bold). On requests to GVP 4-7, this entry can be used to interpolate the desired SPP; thus, speculative TLBs achieve higher capacity.

Overall, GLUE achieves performance benefits by leveraging *one-dimensional* large pages in the guest to approach the benefits of true, unsplintered, two-dimensional large pages. Speculation removes two-dimensional page table walks from a program's critical path, boosting performance. All enhancements are hardware-only so that any software and hypervisor may exploit them transparently, making it robust across the range of real-world splintering scenarios.

GLUE's relationship with prior work: GLUE's mechanisms are partly inspired by SpecTLB [15], which speculatively interpolates physical pages for operating systems using reservation-based large pages. Our work differs in a few key ways. First, the

SpecTLB work discusses the possibility of using TLB speculation to improve TLB miss latencies when guests and hosts use large pages, and to let hypervisors use very large (e.g., 1GB) pages while allowing guests to trap to special regions of memory for I/O. We are the first, however, to identify page splintering as a serious problem – and the first to propose and implement interpolation-based TLB speculation hardware mitigate page splintering. Second, we do not require separate, dedicated hardware structures for TLB speculation, unlike SpecTLB (although we propose some small modifications to existing structures). GLUE distinguishes baseline PTE entries, regular large page PTE entries, and speculative large page PTE entries. Third, we go beyond SpecTLB to propose enhancements to basic TLB speculation that mitigate performance degradation from incorrect speculation (e.g., pipeline flush and refetch) via intelligent PTE prefetching, which reduce the need to perform page table walks to verify a subset of the speculations.

In addition, GLUE is also related to recent work on gap-tolerant sequential mappings (GTSM) which promotes superpages in non-contiguous memory [30]. Though there are some similarities with our work, GTSM is applicable to each dimension separately, unlike GLUE. Furthermore, GTSM requires complex software changes for new page table structures; whereas GLUE supports existing software. Further, GTSM's new page table format is unsuitable for one of the most important sources of page splintering – page sharing: Shared pages can be located anywhere in system memory, but GTSM only handles holes within 4MB memory regions.

4.4 Sources of Page Splintering

Despite their ability to lower address translation overheads, large pages impede other lightweight memory management operations. We discuss some of these reasons below and their influence on page splintering.

Page sharing: Memory deduplication or page sharing, an important source of page splintering, is implemented in commercial hypervisors (e.g., ESX, Xen, KVM) and is used with containers (LXC, Docker) to consolidate as many virtual machines as possible on the same physical resources [13, 38, 91]. Memory deduplication requires software to

scan physical memory to identify memory pages with the same content and eliminate redundant copies. While effective at increasing consolidation ratios, our experiments and past work has found that page sharing leads to aggressive page splintering for two reasons [38]. First, baseline pages are much more likely to have equivalent content than large pages. We ran experiments to compare deduplication opportunities when considering baseline 4KB versus large 2MB pages. We found that using small 4KB pages allowed us to deduplicate 4-10× more physical memory than when using large pages, corroborating past results [38]. Second, the overheads of performing a word-by-word comparison for a 2MB page is much higher than that for a smaller, 4KB page [13]. Hence, when consolidation is targeted, large pages are rapidly splintered.

Hypervisors like ESX and KVM use page sharing aggressively both between and within VMs. This is particularly useful in real-world cloud deployments like Amazon's EC2, Oracle Cloud, and IBM Softlayer, which provide technical support for a limited number of OSes (e.g., Linux, Windows, and Solaris) commonly running VMs from one "template" [78]. In these environments, the workloads have great scope for memory deduplication. In fact, recent industry research has noted this trend and is advocating proactive and aggressive splintering of large pages in anticipation of the need for page sharing [38]. While effective at boosting consolidation, address translation performance is sacrificed.

Non-uniform memory: Past work has shown that large pages may fail to deliver benefits, and can actually degrade performance, on today's multi-socket, non-uniform memory access systems (NUMA) [36]. Because NUMA memory is spread across several physical nodes, large pages may contribute to imbalance in the distribution of memory controller requests, reduce locality of accesses, and increase memory latencies. The OS may therefore splinter large pages, with 4KB chunks migrated among the memory nodes to increase locality [36]. This problem will become even more prevalent with the impending adoption of even more complex, non-uniform heterogeneous memory architectures that balance the access latency, bandwidth, and power needs of heterogeneous processing elements consisting of CPUs, GPUs, and other accelerators [68]. Recent advances in technologies like phase-change, ferroelectric, magnetic, and memristor based

RAM, allied with die stacking [12, 61, 92] suggest that intelligent page placement and movement among multiple memories will determine system performance and energy; page splintering will likely increase in these scenarios considerably.

Working set sampling: Hypervisors typically require some mechanisms to estimate the working set sizes of guest OSs. For example, VMware's ESX uses a statistical sampling approach to estimate virtual machine working set size without guest involvement [88]. For each sampling period, the hypervisor intentionally invalidates several randomly selected guest physical pages and monitors guest accesses to them. After a sampling period (usually a few minutes), the fraction of invalidated pages re-accessed by the guest is checked. This fraction is used to infer a VM's working set size. If a randomly chosen 4KB region falls within a large 2MB page, the hypervisor splinters the large page.

Initially, one may consider "repairing" working set estimation in software. This, however, is difficult; for example, one might use dirty and access bits in both guest and hypervisor page tables to detect page accesses instead of invalidating whole translations. Unfortunately, these bits are not supported by, for example, ARM and any Intel chips older than Haswell. In particular, architectures like ARM that implement relaxed memory consistency models struggle to accommodate page table access and dirty bits, which require sequentially consistent reads and writes for correct operation [73]. In addition, our conversations with hypervisor vendors like VMware suggest that they quite hesitant to implement software modules that can only be used on specific architectures supporting these bits.

Live VM migration: This refers to the process of moving a running virtual machine between different physical machines without disconnecting the client or application. Hypervisors typically splinter all large pages into baseline pages in preparation for live migration to identify pages being written to at 4KB granularity. Once memory state has been shifted to the destination, the splintered pages are typically reconstituted into large pages. However, practically, splintering may remain at the destination node, especially if unfragmented free physical memory space to accommodate large pages is

scarce there.

Limited support for large pages: Large pages require hardware and software support. In practice, many systems lack this support in some way. For example, hypervisors may splinter large pages because of limited-capacity, large-page TLBs. Specifically, if an application's working set is scattered over a wide address space range, large page TLB thrashing can occur [16, 86]. System administrators may therefore disable the hypervisor's ability to back guest large pages [86].

In general, while large pages mitigate address translation overheads, they preclude many memory management techniques for large-scale software systems. Fundamental to this tradeoff is the fact that a large page essentially provides a coarse granularity of memory management and monitoring; while it reduces metadata in the form of the number of translation entries needed, it also greatly reduces the effectiveness of operations like page sharing and memory monitoring.

4.5 GLUE Microarchitecture

This section details hardware for interpolation-based TLB speculation. We begin by describing our baseline TLB organization,[1] and how speculation can be overlaid on it. We then describe speculation details and hardware tradeoffs.

4.5.1 TLB Organization

We assume a processor organization that includes per-core two-level TLB hierarchies, as is typical in modern processors [16, 20, 21]. On a memory reference, two L1 TLBs are looked up in parallel, one devoted to 4KB PTEs and another to 2MB PTEs. L1 misses[2] prompt a lookup in the L2 TLB. GLUE detects guest large pages splintered by the hypervisor. Ordinarily, each such page's 512 4KB PTEs are placed in the 4KB L1 and L2 TLBs. GLUE, however, creates a *speculative 2MB entry for the large page in*

[1] Our proposal does not depend on this specific TLB organization, but we describe it to provide a concrete example to explain our technique.

[2] As this paper is about TLBs, for brevity we use the term "L1" to refer to the L1 TLB, and *not* the IL1 or DL1 cache (and similarly for "L2").

one dimension, with two approaches:

L1-only speculation: Here, speculative 2MB entries are placed only in the 2MB L1 TLB, permitting speculation only at the first-level of the TLB hierarchy. This is a minimally-intrusive design as the 4KB L1 and L2 TLBs are left untouched.

L1-L2 speculation: Though L1-only speculation is effective, it can place a heavy burden on the 2MB L1 TLB, which must now cache PTEs for not only two-dimensional, unsplintered large pages, but also for speculative one-dimensional large pages (which, our experiments reveal, there are many of). Therefore, we also study the benefits of placing speculative 2MB entries in *both* the 2MB L1 TLB and the L2 TLB.

In order to cache speculative 2MB entries, the L2 TLB must support multiple page sizes concurrently, but this is not a problem as modern processors already have this [49]. In general, there are many ways to accommodate concurrent page sizes in TLBs, such as skew-associativity [65, 75, 79, 80, 82] and hash-rehashing (column-associativity) [6]. We have evaluated GLUE on L2 TLBs with both schemes; because the performance numbers are largely unchanged, we present results from skew-associative L2 TLBs in this paper. We do note, however, that GLUE merely leverages already existing techniques to support multiple page sizes concurrently in the TLB, and is not particularly reliant on skewing or hash-rehash.

GLUE represents a significant departure from prior work on SpecTLB [15] as we overlay speculation on existing TLB hardware rather than proposing separate structures for speculation.

4.5.2 Speculative TLB Entries

Figure 4.4 shows a speculated 2MB entry in the L1 2MB TLB. Its structure is identical to a standard 2MB entry, with only a `Spec` bit added to distinguish speculated 2MB entries from standard 2MB entries (necessary to ensure that the L2 TLB and page table walker are probed to verify speculation correctness). This bit represents a minor overhead over the ∼60 bits used per 2MB TLB entry. L2 TLB entries are also minimally changed to support speculation. Once again, a `Spec` bit is required to identify

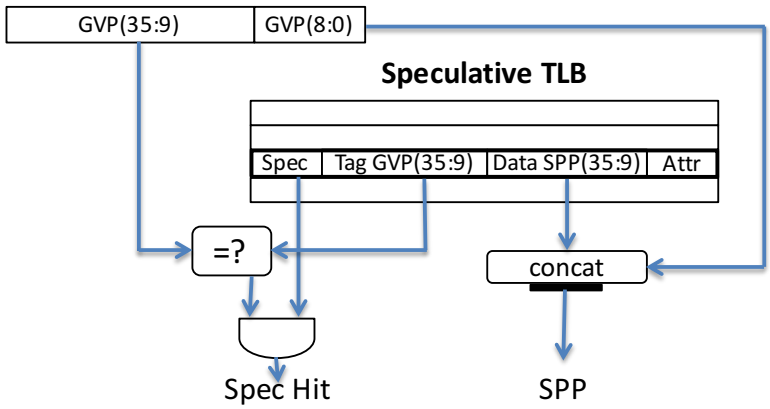


Figure 4.4: Lookup operation on a speculated TLB entry. A tag match is performed on the bits corresponding to its 2MB frame. On a hit, the 2MB frame in system physical memory is concatenated with the 4KB offset within it.

speculated 2MB entries.

4.5.3 TLB Operations

Lookups: Figure 4.4 shows how GLUE performs a speculative lookup for 2MB entries. The guest virtual address is split into a page number and a page offset (not shown). The GVP is further split into a field for 2MB frames (bits 35:9) and the 4KB offset within the 2MB frames (bits 8:0). A lookup compares the 2MB frame bits with the TLB entry's tag bits. A *speculative hit* occurs when there is a match and the `Spec` bit is set. The matching TLB entry maintains the system physical page of the 2MB speculated frame (Data SPP(35:9) in the diagram). This value is interpolated by concatenating the TLB entry data field with the GVP's within-2MB frame offset (GVP(8:0)). The full system physical address is calculated, as usual, by concatenating the guest virtual page offset bits (bits 11:0) with the speculative SPP. This value is then returned to the CPU, which can continue execution while the speculated value is verified. Speculative lookups therefore require minimal additional logic, with only the `Spec` bit check and concatenation operation to generate the SPP.

Fills: GLUE fills speculated 2MB entries into the L1 and L2 TLBs after a page table walk. Suppose a GVP request misses in both L1 and L2 TLBs. The hardware page

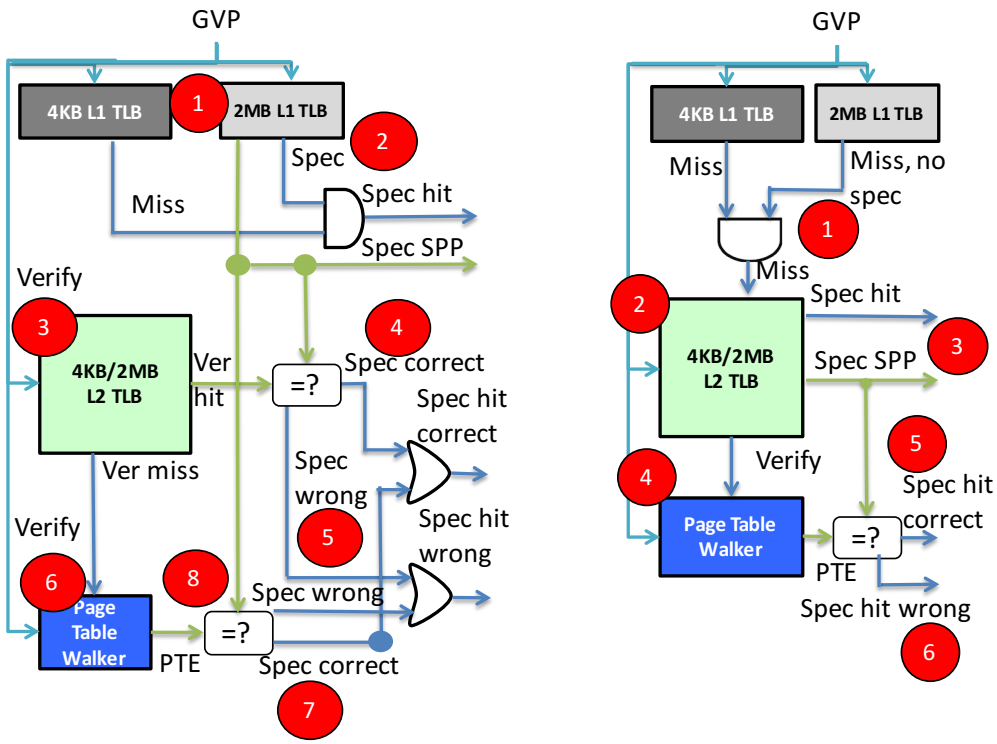


Figure 4.5: The mechanics of TLB speculation. We show the case when (a) we speculate from the 2MB L1 TLB, and (b) we speculate from the L2 TLB.

table walker then traverses both guest and hypervisor page tables, and identifies 4KB GVPs that map to a large guest page but small hypervisor page. These PTEs can be identified from already-existing information on page size in the page tables. For these GVPs, the speculated 2MB frame is calculated by dropping the bottom 9 bits from the corresponding SPP. Then, the PTE for the requested 4KB GVP is placed into the 4KB TLB (as usual), while the speculated 2MB entry is also placed into the 2MB L1 TLB and L2 TLB. Therefore, identifying one-dimensional large pages requires *no additional hardware* beyond standard page table walks.

4.5.4 Speculation Details

We now study various aspects of GLUE, focusing on L1-L2 speculation as it is a superset of L1-only speculation. Figure 4.5 details the control and data flow through the TLBs

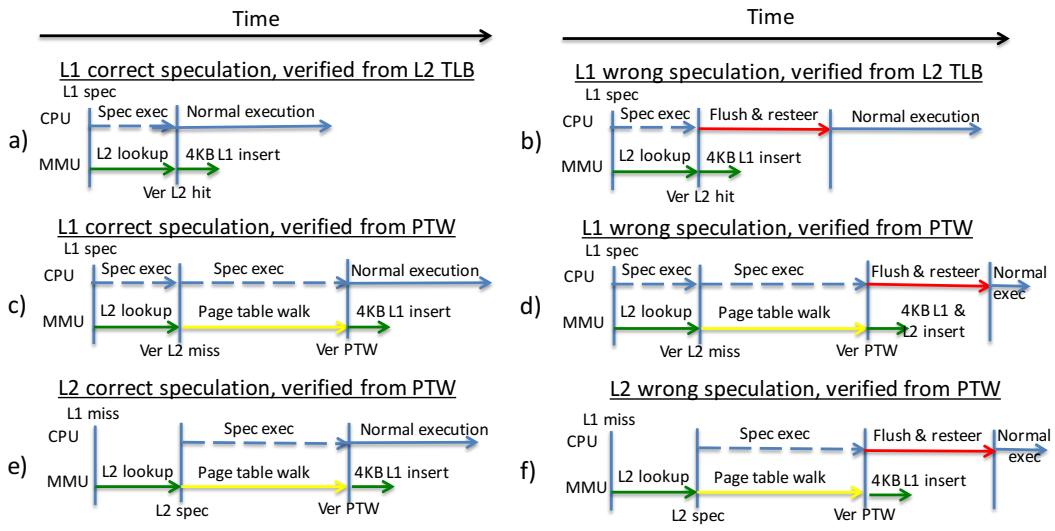


Figure 4.6: Timelines for (a) speculating from the 2MB L1 TLB correctly, and verifying this in the L2 TLB; (b) mis-speculating from the 2MB L1 TLB, and verifying this in the L2 TLB; (c) speculating from the 2MB L1 TLB correctly, and verifying with a page table walk; (d) mis-speculating from the 2MB L1 TLB, and verifying with a page table walk; (e) speculating from the L2 TLB correctly, and verifying with a page table walk; and (f) mis-speculating from the L2 TLB, and verifying with a page table walk.

to support GLUE. Figure 4.6 illustrates the timing of events corresponding to different hit/miss and speculation scenarios.

Correct L1 speculation, verified in the L2 TLB: Figure 4.6(a) illustrates the case where GLUE speculates correctly from the L1 TLB and completes verification from the L2 TLB. The CPU first checks the two L1 TLBs (4KB and 2MB) ①; it misses in the 4KB TLB, finds a speculative entry in the 2MB TLB that results in a speculation hit ②. The hit signal and corresponding speculated SPP are sent to the CPU, which can continue execution while the speculated SPP is verified in parallel. Verification proceeds by checking the L2 TLB ③, which produces a hit on either a matching 4KB entry or a clustered bitmap corresponding to the speculative 2MB entry ④ (speculation confirmed). The 4KB entry is then installed into the L1 4KB TLB, but this occurs off of the critical path.

Incorrect L1 speculation, verified in the L2 TLB: This case starts out the same as in the previous scenario, but when we hit in the L2 TLB, we discover that the actual

SPP is not the same as the interpolated SPP ⑤. This triggers a pipeline flush and refetch as any consumers of the load may have started executing with an incorrectly loaded value. Figure 4.6(b) shows the corresponding timing of events.

Correct L1 speculation, verified by a page table walk: This case is also similar to the first scenario, except that when the L2 TLB lookup is performed, no matching entry is found. A page table walk retrieves the correct translation ⑥, which is found to be the same as the speculated SPP. As shown in Figure 4.6(c), the page table walk to verify the SPP occurs in parallel with the processor pipeline's continued execution using the speculative SPP. In this case, the speculation is correct and so the processor was able to run ahead.

Incorrect L1 speculation, verified by a page table walk: This case is similar to the immediate previous one, except that at the conclusion of the page table walk ⑥, the correct SPP is found to differ from the interpolated SPP. The processor initiates a pipeline flush and refetch. For this case, we also insert the 4KB translation into both L1 and L2 TLBs. The L1 insertion attempts to avoid speculation, and the L2 insertion attempts to ensure a faster verification process in the event that we speculate again from the L1 2MB TLB.

L2 speculation: Figure 4.5(b) shows the cases where a speculative entry is found in the L2TLB, while Figure 4.6(e) and (f) show the corresponding timeline. These cases parallel the L1 speculation scenarios with the only difference that the lookup misses in all L1 TLBs and the verification (whether correct or not) is performed by a page table walk.

4.5.5 Mitigating Verification Costs

Going beyond prior work on TLB speculation [15], we conduct an in-depth study on the tradeoff between capacity and verification costs of speculative large page translations. We study two questions, in particular.

What should we do with requested baseline PTEs if we were able to correctly speculate on them? Consider the case where we use a 2MB speculative entry to

correctly ascertain 4KB PTEs. After verification, we can either place the 4KB PTE in the TLBs, or not insert it into the TLB hierarchy at all. Both approaches have merit; insertion into the TLBs reduces the cost of verification. Because these PTEs are likely to be used in the near future, placing them in the L1 TLB obviates the need for verification-induced L2 TLB lookups, and possibly full-blown page table walks (on L2 TLB misses). On the other hand, insertion into the TLBs lowers effective capacity; true large page entries replace multiple base page translations (e.g., a 2MB x86-64 PTE covers 512 4KB PTEs).

We have compared inserting non-speculative 4KB PTEs into the L1 TLB (reducing verification of future accesses to the same entry but penalizing limited L1 TLB capacity), and into the L2 TLB (promoting speculation with fast verification from the L2 TLB while saving L1 TLB capacity). We also consider non-desirable extremes where we do not insert the non-speculative 4KB entry into the TLBs at all (maximizing capacity but severely exacerbating verification costs), and where we add the non-speculative 4KB PTE in both TLBs (minimizing verification but hurting capacity).

In general, we have found (Section 4.7) that insertion into only the L1 TLB performs best because only tens of 4KB pages within 2MB speculated regions are typically used in temporal proximity. This means that capacity requirements may be relaxed in favor of minimizing the time taken to verify that speculation was correct. Given verification energy requirements and potential performance loss (from greater L2 TLB port contention and cache lookups for page table walks), we believe this is a suitable compromise.

Can we use the extra bits in a speculative large page entry in the L2 TLB to reduce verification costs? Because translations for large pages use fewer bits to represent virtual and physical page numbers, speculative large-page entries maintain many unused bits in a monolithic L2 TLB provisioned to concurrently handle multiple page sizes. In our example, the 2MB speculative large page entries have 18 unused bits. We investigate mechanisms to repurpose these 18 bits to reduce verification costs. Figure 4.7 shows how we use these bits to maintain bitmaps that tell us if 4KB PTEs

within a 2MB speculative large page have the contiguity and alignment to permit correct speculations. For example, Figure 4.7(a) shows the CPU making a request for the translation for VPN 4. A speculative large page entry containing this VPN is discovered in the 2MB L1 TLB, and an interpolated, speculative PPN is sent to the CPU ①. While the CPU continues its operation, the speculative PPN must be verified. The L2 TLB is therefore probed, and the corresponding 2MB speculative translation is located ②. This translation is similar to its counterpart in the L1 TLB but also uses its spare bits to record information about a cluster of 4KB PTEs surrounding the most-recently accessed translation in this speculative 2MB region. We use the unused bits to maintain a cluster number, and a bitmap indicating which PTEs in this cluster are aligned and contiguous (in our page table, all translations aside from the one for VPN 7 are so). Initially this bitmap is empty, therefore it is not able to tell us whether our speculation is correct and we have to walk the page table to verify this ③. We then load the most recently-accessed cluster's information (cluster 0, which VPN 4 falls in) ④. Figure 4.7(b) shows that loading this bitmap in the L2 TLB allows us, in the future when the CPU speculates a PPN from the L1 2MB TLB for VPN 6 ⑤, to quickly verify that this is indeed a correct speculation without the need for an expensive page table walk ⑥. Using the same bitmap, we can also verify quickly if the speculation is incorrect when we speculate a PPN from the L1 2MB TLB for VPN 7.

Through our detailed experiments, we have found that using the 18 spare bits to store two clusters of 8 bits eliminates verification with only 2 additional bits per TLB entry.Furthermore, since typical 64-byte cache lines maintain eight 8-byte PTEs, we use simple combinational logic proposed in chapters 2 and 3 to set the bitmap after a page table walk *without any additional memory references for the walk.*

4.5.6 Mitigating Mis-speculation Overheads

Our real-system measurements showed that loads that miss in the TLBs typically then also miss in the cache hierarchy (90% go to the L3, and 55% to main memory). This is intuitive because PTEs that miss in the TLBs have not been accessed recently; therefore the memory pages they point to are also likely cold. At the point of misspeculation

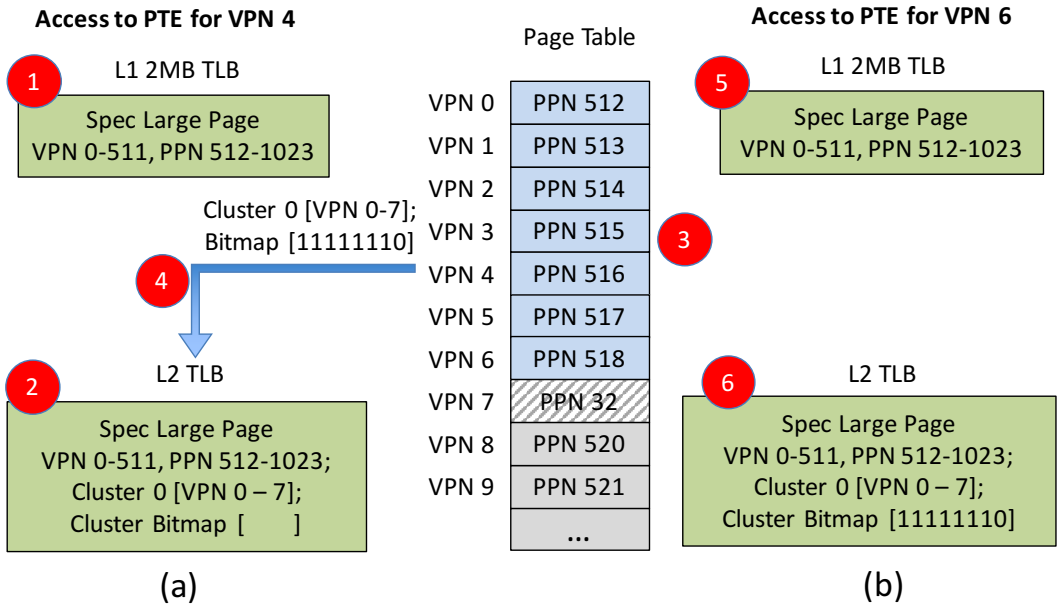


Figure 4.7: Storing clusters of bits (in otherwise wasted L2 TLB entry bits) to eliminate the need for verification-induced page table walks.

detection, the SPP is now known and a prefetch of the address into the DL1 can be started in parallel with the pipeline flush process. For our workloads, it turns out that the TLB speculation rate is sufficiently high such that mitigating the rare flushing event has only a minor impact on overall performance, but this mechanism can provide a more robust solution in the face of less speculation-friendly workloads.

4.6 Experimental Methodology

To evaluate functional, behavioral, and performance effects, we examine systems running multiple real OSs and hypervisors. Our proposal also includes microarchitectural modifications that cannot be evaluated on real systems, while current cycle-level simulators do not support multiple VMs and OSes in full-system simulation modes. For these reasons, like most recent work on TLBs [15, 16, 20, 34, 69, 70], we use a combination of techniques including tracing and performance counter measurements on real machines, functional cache-hierarchy and TLB simulators, and analytical modeling to estimate the overall impact on program execution time.

4.6.1 Workloads

We set up our virtualization environment on a host machine with 8 CPUs and 24GB RAM. We deploy 8 VMs, each has 3GB of RAM for BioBench and SPECcpu workloads, and 4VMs, each with 4GB of RAM for Cloudsuite workloads. The host uses VMware ESXi server to manage VMs. All VMs have Ubuntu 12.04 server, and large pages are enabled using Transparent Hugepage Support (THS) [10]. In addition, to showcase the generality of our observations across hypervisors and architectures, we evaluate KVM on the same hardware and KVM on an ARM system with four Cortex A15 cores. Finally, we use perfmon2 to read performance counter values in the VMs. We use a wide set of benchmarks, from SPECcpu 2006, BioBench [8], and CloudSuite [32] that have non-negligible TLB miss overheads. We present results on workloads sensitive to TLB activity. For our container-based studies, we use linux containers (LXC) with KSM [11].

4.6.2 Trace Collection

We use Pin [58] to collect guest memory traces for our workloads. The original pintool only provides virtual addresses, hence we extend the Linux pagemap to include physical addresses and intermediate page table entries (PTE) to be read by our pintool. For each workload, we select a PinPoint region of one billion instructions [66], and we validate the MPKI of the trace with performance counter measurements to ensure that the sampled region is representative of the benchmark.

We use VMware VProbes scripts [87] to collect hypervisor memory traces, which contain guest and system physical addresses. We rely on guest physical addresses, which are seen in both guest and hypervisor traces to get a complete trace of guest virtual, guest physical, and system physical addresses for our simulator. We also extend the tracing utility to VMs on KVM hypervisor to get similar information.

4.6.3 Functional simulator

To determine the hit-rate impact of the different TLB structures, we make use of a functional simulator that models multi-level TLBs, the hardware page-table walker, and the conventional cache hierarchy. The TLBs include a 64-entry, 4-way DTLB for 4KB pages; a 32-entry, fully-associative DTLB for 2MB pages; and a 512-entry, 4-way level-two TLB (L2TLB) with concurrent support for 4KB and 2MB pages, similar to Intel's Haswell cores. Our L2 TLB uses a skewed-associative organization [80] for multiple page size support (we have also modeled hash-rehash approaches, which negligibly changes performance benefits). The modeled TLB hierarchy also includes page walk caches that can accelerate the TLB miss latency by caching intermediate entries of a nested page table walk [14, 20]. The simulator has a three-level cache hierarchy (32KB, 8-way DL1; 256KB, 8-way L2; 8MB, 16-way L3 with stride prefetcher), which the modeled hardware page table walker uses on TLB misses.

4.6.4 Analytical Performance Model

For each application, we use the real-system performance counter measurements (on full-program executions) to determine the total number of cycles `CPU_CYCLES` (execution time), the total number of page-walk cycles `PWC` (translation overhead, not including TLB access latencies), the number of DTLB misses that resulted in L2TLB hits, and the total number of L2TLB misses (which then require page table walks). In addition, we also make use of several fixed hardware parameters including the penalty to flush and refill the processor pipeline (20 cycles, taken to be about the same as a branch misprediction penalty [4]), the DTLB hit latency (1 cycle), and the L2TLB hit latency (7 cycles).

The analytical performance model is conceptually simple, although it contains many terms to capture all of the different hit/miss and correct/wrong speculation scenarios covered earlier in Figure 4.6. In the baseline case without GLUE, the program execution time is simply `CPU_CYCLES`. From the performance counters, we can determine the total number of cycles spent on address translation overheads `ATO` (e.g., page table walks),

and therefore `CPU_CYCLES - ATO` gives us the number of cycles that the processor is doing "real" execution `BASE_CYCLES` (i.e., everything else but address translations). In other words, this would be the execution time if virtual memory was completely free. From here, our analytical model effectively consists of:

$$\texttt{Execution\_Time} = \texttt{BASE\_CYCLES} + \sum_i \texttt{ATO}_i \tag{4.1}$$

where each $\texttt{ATO}_i$ is the number of cycles required for address translation for each of GLUE's hit/miss and speculation/misspeculation scenarios.

For example, consider when we have a DTLB miss but we find a speculative entry in the 2MB TLB *and* the speculative translation turns out to be correct, then the number of cycles for address translation would simply be the latency of the L1 TLB (both 4KB and 2MB TLBs have the same latency in our model), as we assume that the verification of the speculation can occur off of the critical path of execution. Our functional simulation determines how often this happens in the simulated one-billion instruction trace, we linearly extrapolate this to the full-program execution to estimate the total number of such events[3], and multiply this by the L1 TLB latency to determine the execution cycles due to this scenario.

For a slightly more interesting example, consider the case shown in Figure 4.6(d) where we miss in the L1 4KB TLB, find a speculative entry in the L1 2MB TLB, the speculation turns out to be wrong, but it required a full page-table walk to determine this (i.e., the PTE was not in the L2TLB). The number of cycles for such an event is:

$$\texttt{L2TLB\_LAT} + \texttt{PW\_LAT} + \max(\texttt{DATA\_LAT}, \texttt{BMP}) \tag{4.2}$$

which corresponds to the L2TLB access latency (need to perform a lookup even though it misses), the page-walk latency `PW_LAT` (we use the average as determined by performance counters), and then the longer of either the data cache lookup `DATA_LAT` or the branch misprediction penalty `BMP`. Assuming we use the prefetch optimization described in Section 4.5.6, when we detect the misspeculation, we can then concurrently flush the

[3] We are confident in our extrapolation methodology as we have validated the performance projections from our 1B instruction traces against performance counter measurements taken from the corresponding *full-program* executions.

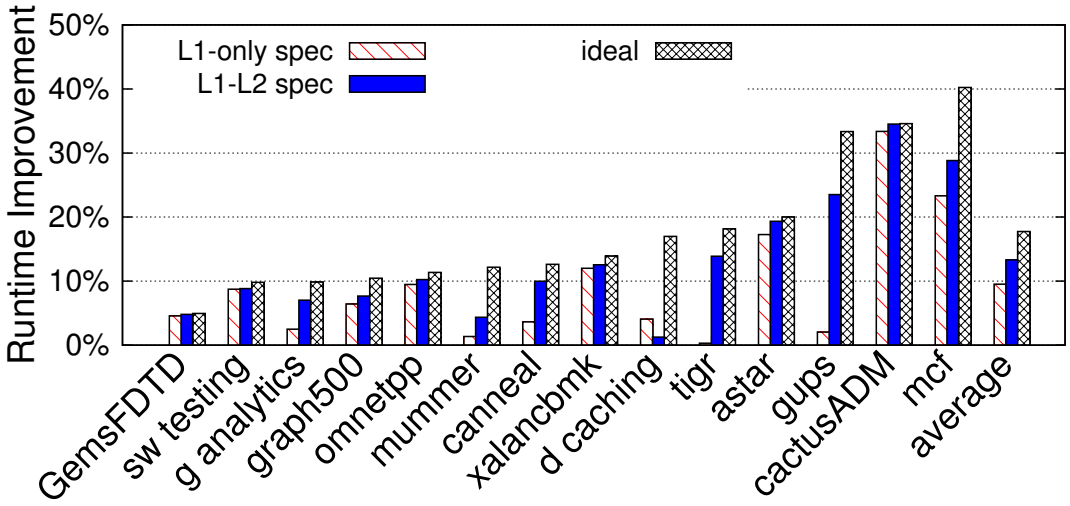


Figure 4.8: Performance benefits of L1-only, L1-L2 speculation, compared to the ideal case without TLB miss overheads. Performance is normalized to the baseline single-VM.

pipeline and start the prefetch of the load into the DL1. Because these events occur in parallel, we take the maximum of these terms. Due to space constraints, we omit explanations for the remaining `ATO_`i equations, but they all follow a similar form that reflects what has already been described in Figure 4.6. It should be noted that such a methodology based on analytically adjusting real-machine measurements has been used in other recent virtual memory research studies [16].

4.7 Experimental Results

4.7.1 GLUE Performance Results: Single VM

We first consider the performance improvements of L1-only and L1-L2 speculation, also showing the importance of careful verification control.

GLUE performance: Figure 4.8 quantifies the benefits of GLUE (all results are normalized to the runtime of the application on a single VM) for L1 and L1-L2 speculation, showing it eliminates the vast majority of TLB overheads in virtualized systems with splintering. On average, runtime is improved by 14%, just 4% away from the performance of an ideal system with no address translation overheads (i.e., there are never any L1 TLB misses). Most benchmarks are actually significantly closer to the ideal case with only `mummer`, `data caching`, `gups`, and `mcf` showing a difference. For `mummer` and

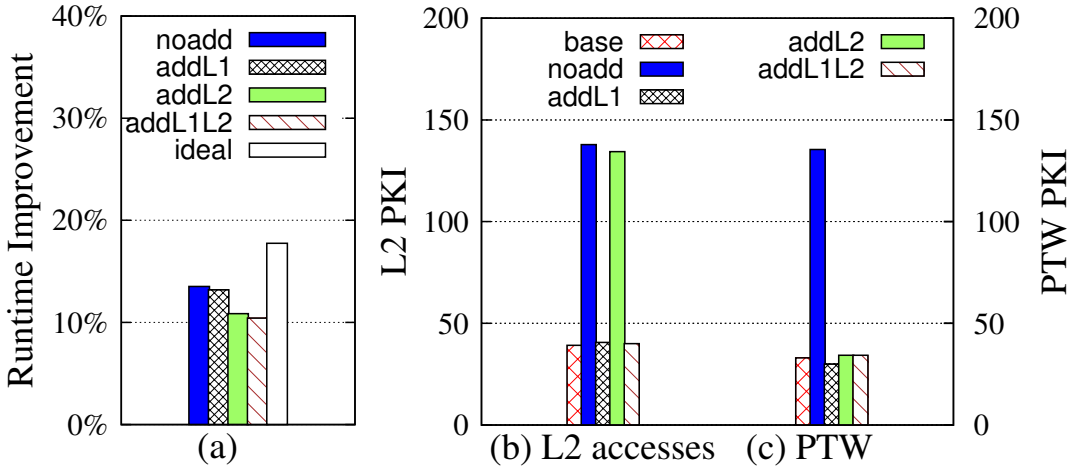


Figure 4.9: Average (a) performance improvements when inserting the non-speculative 4KB PTE, after correct speculation, in neither TLB (noAdd), the L1 TLB (addL1), the L2 TLB (addL2), or both (addL1L2), compared with the ideal improvement; (b) number of L2 TLB accesses per kilo-instruction (APKI) including verification compared to a baseline with speculation; and (c) number of page table walks per kilo-instruction.

`data caching`, this occurs because they are the only benchmarks where the guest generates fewer large pages (see Figure 4.2b); nevertheless, performance benefits are still 5%. For `gups`, and `mcf`, the difference occurs because these benchmarks require more 2MB entries (speculative or otherwise) than the entire TLB hierarchy has available; nevertheless, we still achieve 24%, and 30% performance gains, respectively.

Interestingly, Figure 4.8 also shows that L1-only speculation is highly-effective, achieving 10% performance benefit. In fact, only `mummer`, `tigr`, `gups`, and `graph analytics` see significantly more performance from speculating with both L1 and L2 TLBs.

Mitigating verification costs: To balance the capacity benefits of speculative 2MB entries against the overheads of verification, we insert the non-speculative 4KB PTE corresponding to a correct speculation into the 4KB L1 TLB. Figure 4.9 evaluates this design decision versus a scheme that inserts the non-speculative 4KB PTE into the L2 TLB instead, into both, or into neither. We show the performance implications of these decisions and the number of additional L2 TLB lookups and page table walks they initiate to verify speculations (per kilo-instruction). All results assume L1-L2 speculation; we show average results because the trends are the same across benchmarks.

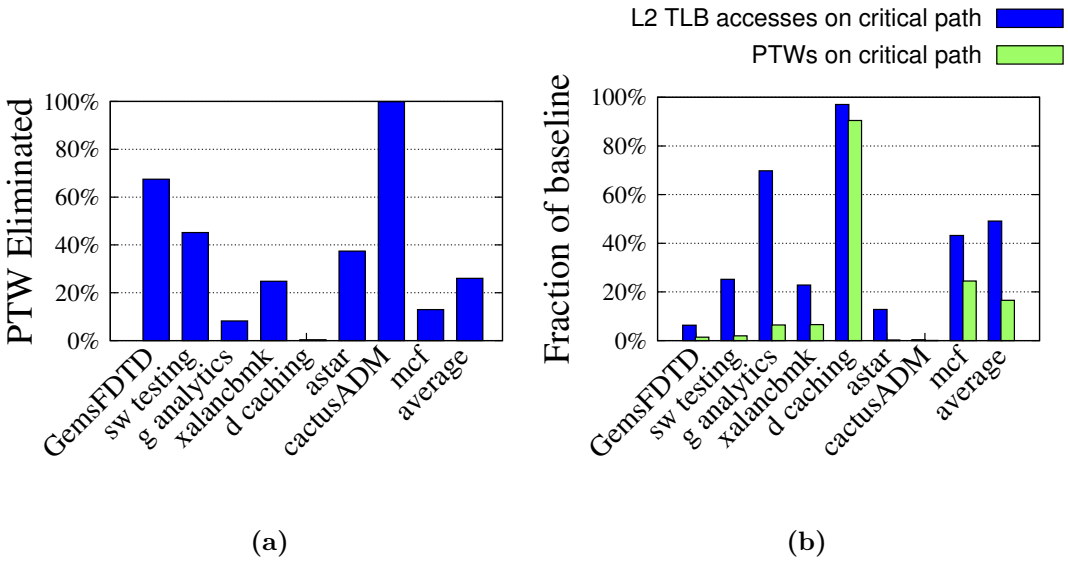


Figure 4.10: (a) Fraction of page table walks eliminated using clustered bitmaps in speculative L2 TLB entries; and (b) fraction of the baseline L2 TLB accesses and page table walks remaining on the critical path of execution with TLB speculation.

Figure 4.9(a) shows that in general, `noAdd` performs the best because a single speculative 2MB entry is used in the entire TLB hierarchy for information about any constituent 4KB SPP. However, inserting non-speculative 4KB PTEs into the L1 TLB (`addL1`), the L2 TLB (`addL2`), or even both (`addL1L2`) performs almost as well (within 2%). Figures 4.9(b)-(c) shows, however, that these schemes have vastly different verification costs, by comparing the additional page walks per kilo-instruction and L2 TLB accesses per kilo-instruction they initiate. Not only does `noAdd` roughly triple the page table walks and L2 TLB accesses, even `addL2` only marginally improves L2 TLB access count. Therefore, we use `addL1` because its verification costs are comparable to the baseline case without sacrificing performance.

Figure 4.10a quantifies the impact of the L2 entry cluster bitmaps. We show the fraction of original page table walks eliminated; on average, 27% of the costly page table walks are eliminated, with absolutely no loss in performance and only 2 additional bits per TLB entry. Some workloads like `cactusADM` are almost entirely freed of page table walks, while others like `GemsFDTD` and `software testing` see 70% and 48% eliminated.

Analyzing TLB miss rates: Figure 4.10b profiles how many of the baseline VM's L2 TLB accesses (caused by L1 TLB misses) and page table walks (caused by L2

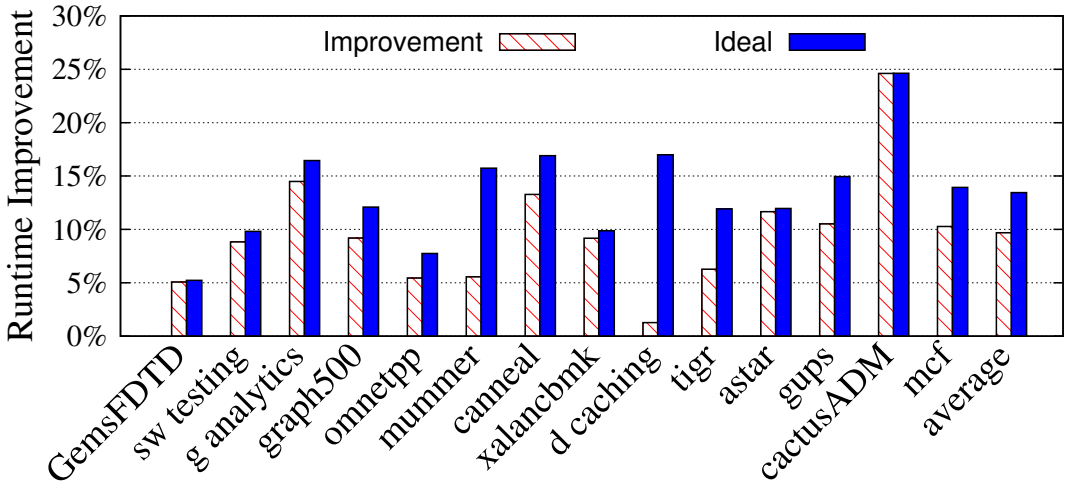


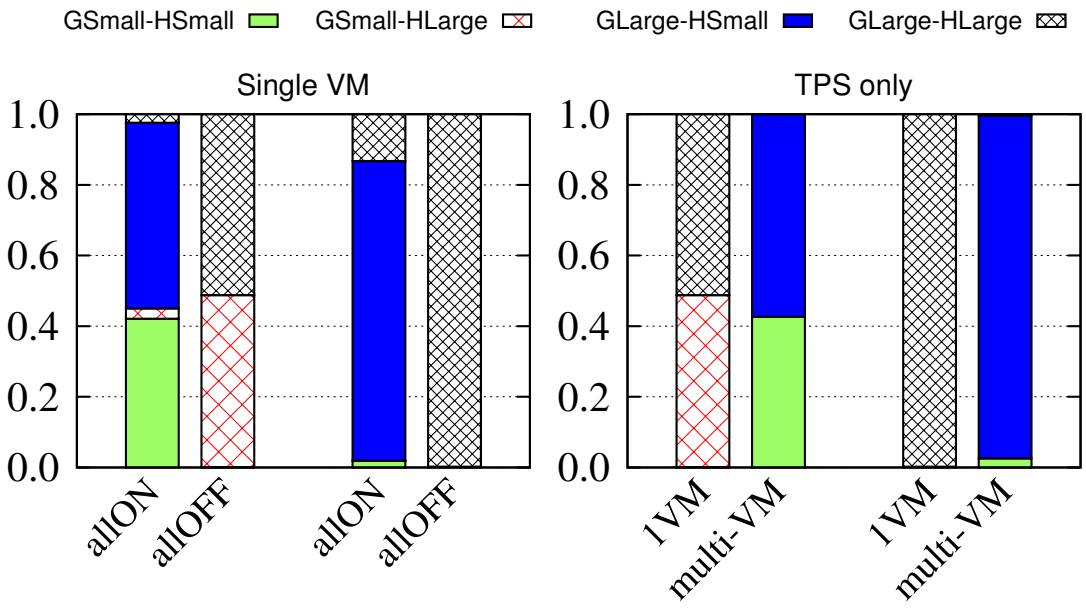
Figure 4.11: Performance gains achieved by GLUE on a multi-VM configuration, compared against the ideal performance improvement where all address translation overheads are eliminated.

TLB misses) remain on the program's critical path. The others are removed from the execution critical path (because they are correctly speculated) and hence do not incur a performance penalty. Overall, Figure 4.10b shows that TLB speculation removes 45% of the L2 TLB lookups and 80% of the page tables walks from the critical path. Some workloads, like `cactusADM`, see almost no page table walks because the clustered bitmap in the L2 TLB completely eliminates verification-induced page table walks (see Figure 4.10a).

4.7.2 GLUE Performance Results: Multiple VMs

VMs with similar workloads: We have investigated the benefits of TLB speculation in scenarios with multiple virtual machines, which may change splintering rates because of inter-VM page sharing, etc. We have studied scenarios with 2, 3, 4, and 8 VMs, but because performance trends are very similar, we show 8-VM results for the SPECcpu and PARSEC workloads. CloudSuite applications, which have far greater memory needs, overcommit system memory with fewer VMs; we hence present 4-VM studies for them.

Figure 4.11 quantifies GLUE's benefits on the multi-VM setup (averaged across VMs, since we find negligible inter-VM variance), compared to an ideal scenario with



(a) mummer canneal **(b) mummer canneal**

Figure 4.12: (a) Effect of page sharing and memory sampling turned on (allOn) in a single VM versus all off (allOff) on page splintering; and (b) Effect of inter-VM page sharing on page splintering in multi-VM settings.

no TLB misses. Multiple VMs stress TLBs even further due to greater contention. Fortunately, sufficient page splintering and good alignment remains, letting GLUE (with L1-L2 speculation) eliminate 75% overheads on average. `GemsFDTD`, `astar`, `cactusADM`, and `software testing` see virtually no address translation overheads. Because real-world virtualization deployments commonly share a single physical node among multiple VMs, GLUE has high real-world utility.

VMs with different workloads: We also study the rarer case where one physical node runs multiple VMs with the same OS (stock Linux) but different workloads. We were surprised to observe that even with different workloads, there is significant inter-VM page sharing, leading to ample page splintering. For example, we found that at least 80% of all TLB misses were to pages that were large in the guest and small in the hypervisor when running VMs with `mcf`, `graph500`, and `gups`. Notable culprits were pages shared from the OS images across VMs and shared libraries. In addition, zero-value pages across VMs were also shared [11]. Fortunately, GLUE counters page sharing-induced splintering even when VMs run different workloads, greatly boosting performance.

4.7.3 Characterizing Page Splintering Sources

By default, ESX uses working set sampling and page sharing for overall performance [13]. Figure 4.12a shows how guests and the hypervisor allocate small and large pages, assuming a single VM running on the physical host. Results are shown for `canneal` (whose behavior almost exactly matches all the other benchmarks and is hence indicative of average behavior) and `mummer`, which illustrates more unique behavior. We compare the default setting against a case where sampling and sharing are turned off. Clearly, turning off sampling and page sharing recovers almost all of the opportunity lost by page splintering.

Figure 4.12b extends these observations when multiple VMs share a physical machine. Because multiple co-located VMs have many pages to share (from the kernel and application), assuming sampling is disabled, page sharing splinters most of the guest large pages.

4.7.4 Importance of GLUE in Future Systems

Our application of GLUE to hypervisor-based virtualization targets scenarios where guests can indeed create large pages, which is purely a function of OS large page support. Modern operating systems have sophisticated and aggressive support for large pages [10,70,83], so guests are likely to continue generating large pages. Nevertheless, we now consider unusual scenarios which could impede guest large page creation.

One might initially consider that memory fragmentation on the guest might curtail large page use in some scenarios. To this, we make two observations. First, VMs are typically used on server and cloud settings to host an isolated, single service or logically related sets of services. It is highly unlikely that fragmentation from competing processes are an issue. Second, in the unusual case where this is an issue, many past studies on large-page support conclude that sophisticated already-existing memory defragmentation and compaction algorithms in OS kernels [10,70] drastically reduce system fragmentation. To test this, we ran our workloads in setups where we artificially fragmented system memory heavily and completely using the random access memhog

process [70]. We found that even for workloads with the largest memory footprints (e.g., `mcf`, `graph500`, `data analytics`, `data caching`, and `software testing`), there were negligible changes in the number of guest large pages allocated, their splintering rates, and how well-aligned the ensuing splintered 4KB pages were. GLUE remains effective in every single, aggressively-fragmented setup that we investigated.

One might also consider the impact of memory ballooning on the guest's ability to generate large pages. Ballooning is a memory reclamation technique used when the hypervisor is running low on memory (possibly in response to the demands of concurrently-running VMs). When the balloon driver is invoked, it identifies 4KB page regions as candidates to relinquish to the hypervisor, and unallocates them. In effect, this fragments the guest's view of physical memory, hampering large page allocation, or breaking already-existing large pages. To study this, we have run several experiments on our setups. Since hypervisors like KVM and Xen expose ballooned pages to the memory defragmentation software run by the kernel, ballooning has no impact on guest large page generation [10].

4.7.5 Understanding GLUE's Limitations

GLUE activates TLB speculation only for memory regions where a large guest page is splintered by the hypervisor and identified by the page table walker as such. Therefore, GLUE is ineffective (though not harmful) when the guest is unable to generate large pages. However, TLB speculation can actually be harmful when the 4KB pages in a speculated large page region are not well-aligned; in these cases, frequent TLB mis-speculations introduce pipeline flushes and refetches, degrading performance. We have not encountered a single case where mis-speculations degrade performance in practice, but we detail how to handle this should it become an issue for other workloads we have not evaluated.

Section 4.5.6 explained that TLB misses are frequently followed by long-latency accesses to the lower-level caches or to main memory to retrieve the requested data. Because L3 caches and main memory typically require 40-200 cycles on modern systems [21, 69, 70], these latencies usually exceed (or are at least comparable) to the cost

of flushing and steering the pipeline on a mis-speculation. Therefore, by initiating a cache prefetch for these data items as soon as a mis-speculation is detected, we can usually overlap mis-speculation penalties with useful work. Because all our real-system configurations enjoy accurate speculation, cache prefetching is not really necessary (on average, we gain roughly 1% more performance). We have calculated the minimum correct speculation rate required to ensure *no performance loss*; for every single benchmark evaluated, 48% speculation accuracy (a pessimistic scenario compared to the 90% accuracy we see in all our configurations) results in no performance degradation.

4.8 Related Work

Beyond recent work on TLB speculation [15], the rising costs of address translation on big-memory systems have prompted researchers to perform many other studies on TLB design [14, 16, 20, 34, 70]. While some of these efforts have focused on mostly hardware efforts [14,20,69,70], others have shown the benefits of efficient OS-hardware co-design [16,30,50]. In particular, recent work on redundant memory mappings [50] has interesting implications on page splintering since it employs eager allocation - depending on workload configuration and hypervisor decision-making, splintering could occur at the granularity of ranges of (possibly large) pages. We will investigate the interplay between our splintering approaches and redundant mappings in future work.

4.9 Summary

This work observes the fundamental conflict between the address translation benefits of large pages versus the desire for finer-grained monitoring and agile memory management. We ask the question: is it possible to provide hardware support that enables us to ally the TLB reach benefits of large pages with memory management issues like lightweight memory monitoring, smoother page sharing, and seamless management with NUMA systems. Our proposed hardware uses interpolation-based TLB speculation to achieve this, boosting hypervisor- and container-based virtualization. Overall, while we observed that splintering is a problem and can cause significant performance

problems, our proposed GLUE architecture can largely mitigate these issues, thereby making virtualized systems more attractive to deploy.

Chapter 5

Conclusion

In this disseration, we explored several novel techniques to reduce majority of the address translation overhead in virtual memory systems. In particular, our techniques took advantage of differences arising from the gap between a rigid hardware memory management unit and rich software components in managing memory.

In chapter 2, we introduced the concept of intermediate contiguity, which is transparently generated by the interactions between program's memory faulting order, memory allocator, and memory compaction. In chapter 3, we extended this concept by considering richer and more prevalent patterns than contiguous locality, namely clustered locality, which allows out-of-order mappings between virtual and physical pages. We proposed CoLT and clustered TLB techniques to exploit these patterns respectively. The combination of these techniques help increase the effective reach of TLBs and reduce almost half of the page walk overhead on average.

In chapter 4, we presented a comprehensive characterization of the prevalence and sources of page splintering, which is a major source of the performance difference between native systems and virtualized systems runtime. We observed that even when page splintering happens, the majority of base pages remain aligned within the boundary of the original large page. In response, we proposed GLUE, a low-complexity speculation hardware that identifies contiguous, aligned, but splintered large page regions. Our design removed TLB misses handling from the processor's critical path, and reduced 77% of the translation overhead in virtualized systems as a result.

Overall, this thesis provides an initial approach to taking advantage of the underlying structure and patterns in page tables while completely maintaining the traditional virtual memory abstractions as far as the software layers are concerned. We believe that

our approach will spur more research to hunt for additional, richer patterns in the virtual memory system, deeper analyses of the underlying application and OS/hypervisor behaviors that lead to these, and ultimately more and better mechanisms to allow physical memory sizes to scale without being hamstrung by the virtual memory system.

References

[1] AMD64 Architecture Programmers Manual. White paper, Advanced Micro Devices Inc., 2013. http://developer.amd.com/.

[2] ARMv8-A Reference Manual. White paper, ARM Holdings, plc, 2013. infocenter.arm.com.

[3] Linux* Containers Streamline Virtualization and Complement Hypervisor-Based Virtual Machines. White paper, Intel Corp., 2014. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/linux-containers-hy

[4] Software Optimization Guide for AMD Family 15h Processors. Technical report, Advanced Micro Devices Inc, 2014.

[5] Intel 64 and IA-32 Architectures Software Developers Manual. White paper, Intel Corp., 2015. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-s

[6] A. Agarwal and S. Pudar. Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches. *ISCA*, 1993.

[7] J. Ahn, S. Jin, and J. Huh. Revisiting Hardware-Assisted Page Table Walks for Virtualized Systems. *ISCA*, 2012.

[8] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Intl. Symp. on Performance Analysis of Systems and Software*, pages 2–9, Austin, TX, March 2005.

[9] AMD Corporation. AMD Programmer's Manual. 2, 2007.

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

[11] A. Arcangeli, I. Eidus, and C. Wright. Increasing Memory Density by Using KSM. *Ottawa Linux Symposium*, 2009.

[12] G. Atwood. Current and Emerging Memory Technology Landscape. *Flash Memory Summit*, 2011.

[13] I. Banerjee, F. Guo, K. Tati, and R. Venkatasubramanian. Memory Overcommittment in the ESX Server. *VMware Technical Journal*, 2013.

[14] T. Barr, A. Cox, and S. Rixner. Translation Caching: Skip, Don't Walk (the Page Table). *ISCA*, 2010.

[15] T. Barr, A. Cox, and S. Rixner. SpecTLB: A Mechanism for Speculative Address Translation. *ISCA*, 2011.

[16] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift. Efficient Virtual Memory for Big Memory Servers. *ISCA*, 2013.

[17] A. Basu, M. Hill, and M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. *ISCA*, 2012.

[18] R. Bedicheck. SimNow: Fast Platform Simulation Purely In Software. *Hot Chips*, 2004.

[19] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. *ASPLOS*, 2008.

[20] A. Bhattacharjee. Large-Reach Memory Management Unit Caches. *MICRO*, 2013.

[21] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared Last-Level TLBs for Chip Multiprocessors. *HPCA*, 2010.

[22] A. Bhattacharjee and M. Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *PACT*, 2009.

[23] A. Bhattacharjee and M. Martonosi. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. *ASPLOS*, 2010.

[24] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Simplications. *IISWC*, 2008.

[25] D. Bovet and M. Cesati. Understanding the Linux Kernel. 2005.

[26] 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*, 2013.

[27] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. Wolf, A. Ghosh, J. Lu, S. Poon, M. Stan, W. Butler, S. Gupta, C. Mewes, T. Mewes, and P. Visscher. Advances and future prospects of spin-transfer torque random access memory. *Magnetics, IEEE Transactions on*, 46(6):1873–1878, June 2010.

[28] J. B. Chen, A. Borg, and N. Jouppi. A Simulation Based Study of TLB Performance. *ISCA*, 1992.

[29] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 28(1):213–216, Mar. 1996.

[30] Y. Du, M. Zhou, B. Childers, D. Mosse, and R. Melhem. Supporting Superpages in Non-Contiguous Physical Memory. *HPCA*, 2015.

[31] Z. Fang, L. Zhang, J. Carter, W. Hsieh, and S. McKee. Reevaluating online superpage promotion with hardware support. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 63–72, 2001.

[32] 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. *ASPLOS*, 2012.

[33] N. Ganapathy and C. Schimmel. General-Purpose Operating System Support for Multiple Page Sizes. *USENIX*, 1998.

[34] J. Gandhi, A. Basu, M. Hill, and M. Swift. Efficient Memory Virtualization. *MICRO*, 2014.

[35] GaTech. Macsim. *http://code.google.com/p/macsim/*.

[36] F. Gaud, B. Lepers, J. Decouchant, J. Funston, and A. Fedorova. Large Pages May be Harmful on NUMA Systems. *USENIX ATC*, 2014.

[37] M. Gorman. Understanding The Linux Virtual Memory Manager. 2004.

[38] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. *VEE*, 2015.

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

[40] HMCC. HMC Specification 1.0, 2013. http: //www.hybridmemorycube.org.

[41] IBM. Big data at the speed of business, 2011. http://www-01.ibm.com/software/data/bigdata/.

[42] Intel Corporation. TLBs, Paging-Structure Caches and their Invalidation. *Intel Technical Report*, 2008.

[43] Intel Corporation. 3D XPoint UnveiledThe Next Breakthrough in Memory Technology, 2015. http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html.

[44] B. L. Jacob and T. N. Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. *SIGPLAN Not.*, 33(11):295–306, Oct. 1998.

[45] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based On-the-fly Multi-core Simulator. *4th Workshop on Modeling, Benchmarking, and Simulation*, 2008.

[46] JEDEC. High Bandwidth Memory (HBM) DRAM (JESD235), 2013.

[47] 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, New York, NY, USA, 2013. ACM.

[48] G. Kandiraju and A. Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-Driven Study. *ISCA*, 2002.

[49] D. Kanter. Haswell Memory Hierarchy. *http://www.realworldtech.com/haswell-cpu/5/*, 2012.

[50] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. Hill, K. McKinley, M. Nemirovsky, M. Swift, and O. Unsal. Redundant Memory Mappings for Fast Access to Large Memories. *ISCA*, 2015.

[51] B. Kero. Running 512 Containers on a Laptop. *http://bke.ro/running-512-containers-on-a-laptop*, 2015.

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

[53] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *36th Intl. Symp. on Computer Architecture*, Austin, TX, June 2009.

[54] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Phase change memory architecture and the quest for scalability. *Commun. ACM*, 53(7):99–106, July 2010.

[55] G. H. Loh and M. D. Hill. Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches. In *MICRO-44*, 2011.

[56] G. H. Loh, N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung. Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems. In *SHAW-3*, 2012.

[57] 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. *PLDI*, 2005.

[58] 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 *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[59] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. 2005.

[60] C. McCurdy, A. Cox, and J. Vetter. Investigating the TLB Behavior of High-End Scientific Appplications on Commodity Multiprocessors. *ISPASS*, 2008.

[61] M. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories. *HPCA*, 2015.

[62] Micron. HMC Gen2, 2013.

[63] R. C. Murphy, K. B. Wheele, B. W. Barrett, and J. A. Ang. Introducing the graph 500. 2010.

[64] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, Transparent Operating System Support for Superpages. *OSDI*, 2002.

[65] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-Based Superpage-Friendly TLB Designs. *HPCA*, 2014.

[66] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *37th Intl. Symp. on Microarchitecture*, Portland, OR, December 2004.

[67] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS*, 2003.

[68] S. Phadke and S. Narayanasamy. MLP Aware Heterogeneous Memory System. *DATE*, 2011.

[69] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing TLB Reach by Exploiting Clustering in Page Translations. *HPCA*, 2014.

[70] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. *MICRO*, 2012.

[71] M. K. Qureshi and G. H. Loh. Fundamental Latency Trade-offs in Architecting DRAM Caches. In *MICRO-45*, 2012.

[72] P. Rogers. AMD heterogeneous Uniform Memory Access. *AMD Whitepaper*, 2013.

[73] B. Romanescu, A. Lebeck, and D. Sorin. Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency. *ASPLOS*, 2010.

[74] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 176–187, New York, NY, USA, 1995. ACM.

[75] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. *MICRO*, 2010.

[76] T. Sato and I. Arita. Low-Cost Value Predictors Using Frequent Value Locality. In *4th Intl. Symp. on High Performance Computing*, pages 106–119, Kansei Science City, Japan, May 2002.

[77] A. Saulsbury, F. Dahlgren, and P. Stenström. Recency-Based TLB Preloading. *ISCA*, 2000.

[78] A. W. Services. AWS Cloud Formation User Guide. 2010.

[79] A. Seznec. A Case for Two-Way Skewed Associative Cache. *ISCA*, 1993.

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

[81] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *MICRO-45*, 2012.

[82] M. Spjuth, M. Karlsson, and E. Hagersten. The Elbow Cache: A Power-Efficient Alternative to Highly Associative Caches. *Uppsala University Technical Report 2003-46*, 2003.

[83] M. Talluri and M. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. *ASPLOS*, 1994.

[84] M. Talluri, S. Kong, M. Hill, and D. Patterson. Tradeoffs in Supporting Two Page Sizes. *ISCA*, 1992.

[85] Virtutech. Simics for Multicore Software. 2007.

[86] VMware. Large Page Performance: ESX Server 3.5 and ESX Server 3i v3.5. *VMware Performance Study*, 2008.

[87] VMware. VProbes Programming Reference. 2008.

[88] C. Waldspurger. Memory Resource Management in VMware ESX Server. *OSDI*, 2002.

[89] H.-S. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. Chen, and M.-J. Tsai. Metal x2013;oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, June 2012.

[90] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. Corner. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. *VEE*, 2009.

[91] J. Xiao, Z. Xu, H. Huang, and H. Wang. Security Implications of Memory Deduplicationi in a Virtualized Environment. *DSN*, 2013.

[92] Y. Xie. Modeling, Architecture, and Applications for Emerging Non-Volatile Memory Technologies. *IEEE Computer Design and Test*, 2011.

[93] Y. Zhang, J. Yang, and R. Gupta. Frequent Value Locality and Value-Centric Data Cache Design. In *9th Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Cambridge, MA, November 2000.