Scalable Virtual Memory via Tailored and Larger Page Sizes

Faruk A. Güvenilir

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2019-001
May 2019

This page is intentionally left blank.

The Dissertation Committee for Faruk Alfredo Güvenilir
certifies that this is the approved version of the following dissertation:

Scalable Virtual Memory via Tailored and Larger Page Sizes

Committee:

Yale N. Patt, Supervisor

Mattan Erez

Mohit Tiwari

Christopher J. Rossbach

Robert S. Chappell

Scalable Virtual Memory via Tailored and Larger Page Sizes

by

Faruk Alfredo Güvenilir

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2019

Dedicated to my brothers and sister.

Acknowledgments

This dissertation could not have been completed without the support of many. Here is my attempt to express my deepest gratitude for the technical, motivational, administrative, and other forms of help and support they graciously contributed throughout my years spent pursuing the Ph.D. at The University of Texas at Austin.

First, I thank my advisor, Yale N. Patt. My time as an undergraduate enrolled in his Introduction to Computing (EE306) course marked the start of my interest in computer architecture. His advice in both technical and non-technical matters helped me grow tremendously. I thank him for his guidance in performing solid research and in effective teaching, writing, and communication. I greatly appreciate the continued support during hard academic times and the patience I was afforded.

I thank Mattan Erez, Mohit Tiwari, Christopher J. Rossbach, and Robert S. Chappell for serving on my dissertation committee and providing me valuable feedback. I also thank Rob for the advice he offered while working as my manager during my several Intel internships and for the final push he gave me to see my Ph.D. through at the conclusion of my summer 2016 internship.

I thank other former and current HPS research group members: M. Aater Suleman, Chang Joo Lee, Marco A. Z. Alves, Carlos Villavieja, Eiman Ebrahimi, Veynu Narasiman, Khubaib, Rustam Miftakhutdinov, José A. Joao, Milad Hashemi, Ben (Ching-Pei) Lin, Stephen Pruett, Siavash Zangeneh Kamali, Ali Fakhrzadeghan, and Aniket Deshmukh. They provided a great research environment, open and candid discussions and feedback on research ideas, and a fun and interesting office to work in. Everyone's willingness to

lend a helping hand, even on non-research related matters, created an exceptionally positive graduate student experience. I additionally specifically thank:

- Aater for discussing graduate school life with me in my final undergraduate semester.
- Eiman for inviting me to contribute to his research my first semester in graduate school, although I was not able to at the time.
- Carlos for the guidance he provided when I was a young graduate student, and for always pushing me to make progress with my research.
- José for his mentorship as a senior student, and for maintaining (and teaching me about) our computer infrastructure prior to my taking over.
- Veynu and Khubaib for being great teaching assistants when I took Introduction to Computing, Computer Architecture, and Microarchitecture.
- Rustam for his extensive contributions to our simulation infrastructure, for prompting me to contribute to his research by getting me involved with the memory scheduler in our research group simulator, and for his mentorship in my later Intel internships.
- Milad for his great work ethic in the various courses we served as teaching assistants or enrolled together in.
- Ben for never failing to inject humor into our work environment, and for being a great partner in several course projects.
- Stephen for his frank feedback on research and other ideas, and for compelling conversations on a variety of topics.

- Siavash for his willingness to help out with our computer infrastructure, and for demonstrating great patience in multiple situations.

- Ali for being a great roommate.

I also thank the administrative staff in the ECE department for their support. I thank Leticia Lira for her outstanding administrative support to the research group and her responsiveness to my many requests over the years. I thank Melanie Gulick for her help with navigating the bureaucratic processes of the department and graduate program. I thank Gabriel Hernandez and Rick Gomez for being outstanding IT staff with the department, always willing to accommodate our research group's special computing needs.

I had the opportunity to intern over several summers with great people at Intel (five times), Apple, and Google. I thank my many coworkers and mentors, including Tse-Yu Yeh, Mark Dechene, and others, for the learning experiences they provided at those companies.

I also thank the many ECE department graduate students of my cohort. There are simply too many to name here.

I thank the many friends who have remained close, some for more than ten years, and others for much fewer (but nevertheless precious) years. I am very grateful to Jeff and Karen Chien, Ran Duan, Scott Liu, Wei Liang, Alex Tang, Scott Chu, Karthik Kolavasi, and many, many others too numerous to name here. Whether it was conversing late into the night on meaningful or entertaining topics, traveling around the country and world on various trips, or meeting up for a simple weekend lunch, their friendship and support helped ensure I was always pursuing my true self.

I thank my family, including my extended family members in Venezuela and Turkey, for their unconditional love and support. I thank my grandparents, José Antonio and Angela Ramona Albornoz. My grandfather passed away as I was completing this dissertation. The year I spent in Venezuela with them when in first grade, and the year they spent with us in my first undergraduate year will always be treasured memories for me. I thank my parents, Abbas and María Soledad Güvenilir. Their efforts and sacrifices have made me who I am today. I thank my mother for simply always being there. I thank my brothers and sister, Abbas Abdullah, José Ali, and Ayşe Angela. Their belief in me and support continues to push me forward. José and Ayşe looking up to me gave me more strength in hard times than they could know. The memories of even simple times spent enjoying a game together as four siblings were enough to motivate me to continue reaching out to the truth. This dissertation would have been impossible without my family's love and support.

Finally, I thank God. In the name of God, the Most Gracious, the Most Merciful.

Faruk Güvenilir

May 2019, Austin, TX

Scalable Virtual Memory via Tailored and Larger Page Sizes

Faruk Alfredo Güvenilir, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Yale N. Patt

Main memory capacity continues to soar, resulting in TLB misses becoming an increasingly significant performance bottleneck. The 4KB default minimum page size in architectures like x86 is decades old and hampers future growth potential. Current coarse grained page sizes, the solution from Intel, ARM, and others, have not helped enough. I propose Tailored Page Sizes on top of a Larger Base Page Size (TPS+). TPS+ allows pages of size 2^n, for all n greater than the minimum page size. TPS+ means one page table entry (PTE) for each large contiguous virtual memory space mapped to an equivalent-sized large contiguous physical frame. To make this work in a clean, seamless way, I suggest small changes to the ISA, the microarchitecture, and the O/S allocation operation. The result: TPS+ can eliminate more than 99% of all TLB misses and page walk memory accesses across a variety of SPEC17 and big data memory intensive benchmarks, yielding 59.7% average performance improvement in virtualized execution scenarios.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

Page based virtual memory has been a fundamental memory-management component of modern computer systems for decades. Virtual memory provides each application with a very large, private virtual address space, resulting in memory protection, improved security due to memory isolation, and the ability to utilize more memory than physically available through paging to secondary storage. In addition, applications do not have to explicitly manage a single shared address space; the virtual-to-physical address mapping is controlled by the operating system and hardware. Current systems divide the virtual address space into coarse-grained, fixed size, virtual pages which are mapped to physical frames via a hierarchy of page tables. For example, x86-64 supports page sizes of 4KB, 2MB, and 1GB. The smallest page size is often referred to as the base page size; larger page sizes are called superpages or huge pages. Translation Lookaside Buffers (TLBs) cache virtual-to-physical translations to eliminate the cost of a page table walk.

Current virtual memory systems have two key limitations not adequately solved by current proposals:

1. Available huge page sizes are at too coarse of a granularity.

2. The base page size of 4KB is decades old and limits scalability.

The current trend of increasing computer system physical memory capacity continues. Client devices with tens of gigabytes of physical memory and servers with terabytes of physical memory are becoming commonplace. Applications that leverage these large physical memory capacities suffer costly virtual-to-physical translation penalties due to realistic constraints on TLB sizes.

At the 4KB page size, a typical L1 TLB capacity of 64 entries will only cover 256 KB of physical memory. This problem is referred to as limited TLB reach. With larger page sizes, current processors typically contain multiple L1 TLBs, one for each supported page size. Even at the 1GB page size, a typical L1 TLB capacity of 4 entries for this page size will only span 4GB of physical memory.

Prior work [8, 14, 24, 41, 42, 44] has demonstrated that some applications can spend up to 50% of their execution time servicing page table walks. Large L2 TLB capacities with thousands of entries reduce some of the impact from infrequent, but still very costly, page walks. Figure 1.1 shows the percentage of total application execution time the processor spends on page walks, as collected from performance counter data on physical hardware with Transparent Huge Pages active. Three cases are shown: 1) native execution with no interference, 2) native execution with a simultaneous multi-threading (SMT) hardware thread competing for TLB resources, and 3) virtualized execution with two dimensional page walks. While native execution page walk overhead is generally modest due to the large L2 TLB capacity, the results show SMT interference and virtualized execution can cause significant increases in page walk overhead. Page walk overhead will further increase with upcoming five level page tables [36].

High numbers of L1 TLB misses can additionally impose a performance penalty. Figure 1.2 demonstrates the performance improvement of a perfect L1 TLB over a perfect

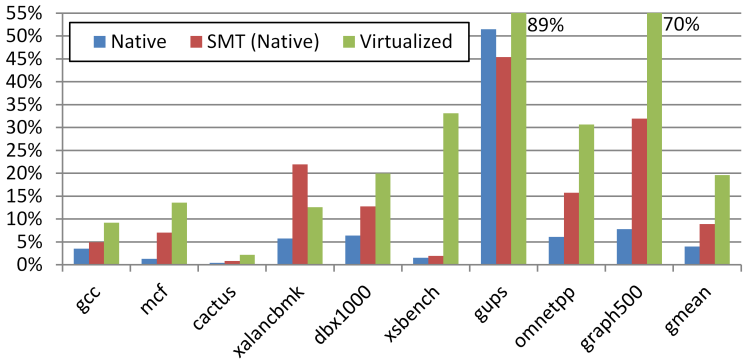

Figure 1.1: Page Walk Overhead: Percent of Execution Time Spent Page Walking

L2 TLB baseline. This study was performed with cycle-based simulation modeling out-of-order effects. The out-of-order window can often hide many L1 TLB misses by overlapping this latency with other useful work. But, when memory accesses are on the critical path of execution (e.g., linked data structure traversal), even frequent L1 TLB misses can cause an appreciable performance penalty, as shown. Both limited L1 and L2 TLB capacity still play major roles in translation overhead.

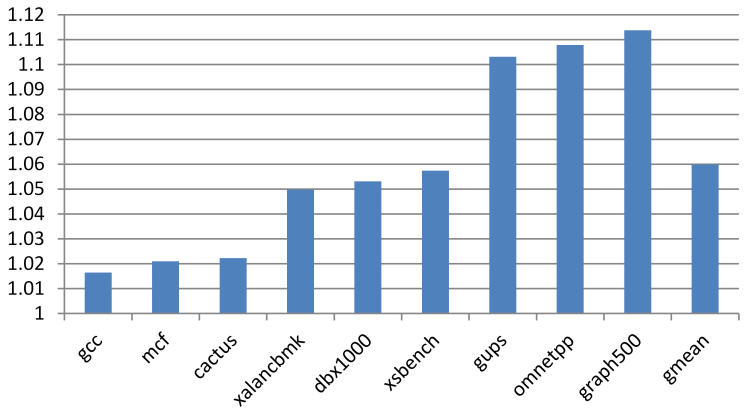

Figure 1.2: L1TLB Miss Overhead: Speedup of Perfect L1 TLB over Perfect L2 TLB Baseline

The existing, small base page size imposes additional restrictions on microarchitectural design. For example, high performance L1 cache designs are limited in capacity to

the page size times the cache associativity.

The coarse granularity of supported page sizes in the most common modern processor families (x86-64 and ARM) are inadequate. For example, consider a single 256 MB data structure. Provided the operating system is able to identify free contiguous memory such that allocating any available page size for this new data structure is possible, the tradeoff between page sizes presents serious problems with any choice. Choosing the 2MB page size requires 128 TLB entries, already well over the L1 TLB capacity for this single data structure. Choosing the 1 GB page size results in 768 MB wasted physical memory, internal fragmentation of 75%. The current trend to increase the gaps between consecutive coarse grained page sizes only worsens the tradeoff. For example, ARM supports a translation granule with coarse-grained page sizes of 64 KB or 512 MB [4].

1.2 The Solution

These problems, along with the continued growth of physical memory capacity indicate change in the virtual memory mechanism is necessary to deal with the growing translation overhead. I propose Tailored Page Sizes (TPS), a simple and clean extension to current processor architectures that reduces translation overhead, significantly lowers page walk memory references, and substantially reduces L1 TLB misses in TLB intensive workloads that will continue to scale with increasing physical memory capacity. TPS introduces support for pages sized at any power-of-two larger or equal to the base page size (e.g., 4KB in x86-64). TPS includes changes to operating system software, the ISA, and the microarchitecture. When fragmentation is high, resulting in insufficient contiguity to utilize coarse-grained pages well, TPS allows the OS to leverage what contiguity it can for performance with intermediate page sizes.

TPS leverages the natural levels of address space contiguity that occur due to the standard operation of the OS buddy allocators and memory compaction daemon. In addition, applications already perform mapping system calls at runtime to large contiguous regions of their virtual address space, but current OS implementations split these mapping requests into however many smaller pages are required to compose the larger request. Utilizing these common properties, TPS is able to create the necessary virtual-to-physical mappings with only a small number of appropriately sized fine-grained pages for many memory intensive applications. Recall the 256 MB data structure: TPS means one appropriately sized page needing only one PTE, as shown in Figure 1.3.

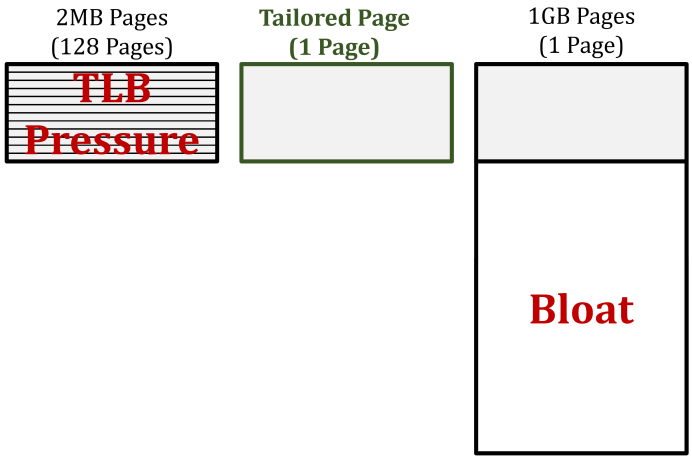

Figure 1.3: TPS: 1 Page, 1 PTE, No Bloat

To facilitate further microarchitectural enhancements, I propose increasing the base page size from 4KB to 256KB. For example, this enables greater freedom in L1 cache design. In addition, the larger page base page size reduces the number of steps in the page walk process, particularly useful for expensive two dimensional page walks on virtual machines. The larger base page size can work in conjunction with TPS (called TPS+) to maximize reduction in translation latency. In isolation, a larger base page size introduces several unaddressed drawbacks; I offer solutions to these challenges.

1.3 Thesis Statement

Architectures leveraging both a larger base page size and application-tailored page sizes can significantly reduce virtual memory and translation overheads, thereby increasing application performance and scalability.

1.4 Contributions

This dissertation makes the following key contributions:

- I design the architectural and minimal operating system features necessary to support application-tailored page sizes at any power-of-two (greater than or equal to the base page size). In contrast to standard coarse grained page sizes, tailored page sizes will enable significant reduction in page walk memory references and TLB misses by presenting the ability to better leverage whatever levels of system physical memory contiguity are present.

- I demonstrate the benefits of increasing the fundamental base page size, such as improved L1 cache designs and reduced worst-case and average-case page walk latency. Increasing the base page size also causes significant drawbacks; I identify solutions to these drawbacks.

- I evaluate via simulation the performance impact of TPS+, and show that TPS+ is able to eliminate nearly all TLB misses and page walk memory references, resulting in 59.7% average performance improvement in virtualized execution scenarios.

1.5 Dissertation Organization

This dissertation is organized into six chapters. Chapter 2 provides background information and describes the most relevant prior work. Chapter 3 presents and evaluates TPS, a proposal to tailor page sizes based on application demand. Chapter 4 addresses the benefits and challenges of increasing the architectural base page size. Chapter 5 describes in detail related work. Chapter 6 concludes the dissertation and identifies future work.

Chapter 2

Background

Virtual memory provides each process with the illusion of a very large, private address space. Hardware performs address translation at runtime, while the operating system (OS) is responsible for partitioning the virtual address space into virtual pages, and mapping them to physical frames. The architecture defines the contract between the hardware and software. The following topics will directly apply to most architectures utilizing a hierarchical page table tree that supports coarse-grained superpages (e.g., ARM, x86-64). I will primarily cover the x86-64 architecture for ease of explanation.

2.1 Virtual Memory Hardware

2.1.1 Page Tables

The page table contains the mappings from virtual page to physical frame for the process address space. The page table itself is stored in memory (and mapped as part of the virtual address space), and consists of page table entries (PTEs). Each PTE contains a single mapping from virtual page to physical frame, as well as bookkeeping bits for protection and other purposes. Intel x86-64 currently implements the page table as a four level hierarchical radix tree. Every memory reference executed by a process uses virtual addresses. The processor core contains a hardware page table walker to start with the original virtual address, and traverse the page table tree to eventually produce the PTE

containing the page frame number for the requested virtual address. Figure 2.1 illustrates the page walk process. Bits 63 to 48 of the virtual address are currently unused.

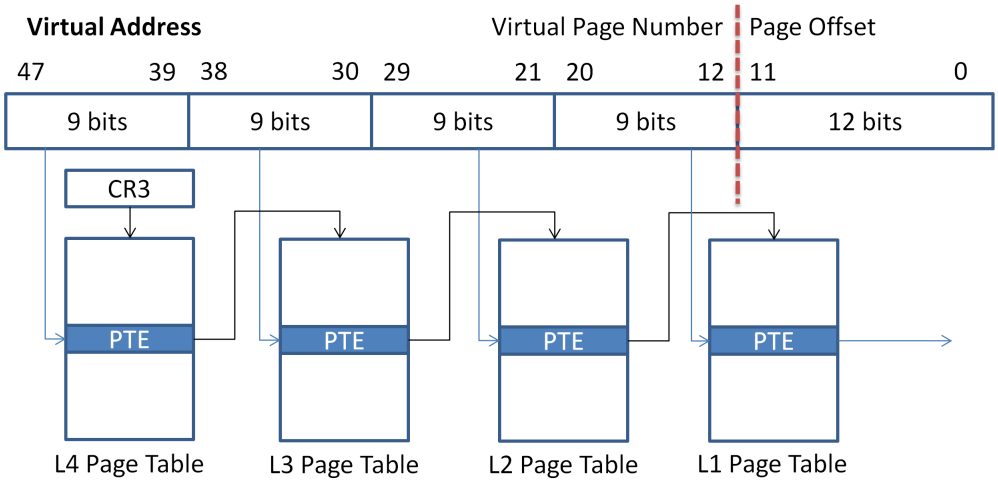

Figure 2.1: Walking the x86-64 Page Tables

Virtual machines consist of guest operating systems (gOS) running on a host operating system (hOS). Addresses must be translated from guest virtual address (gVA) to host virtual address (hVA, also called guest physical address) to machine physical address (mPA, also called host physical address). The standard page tables translate from gVA to hVA. Intel extended page tables (EPT) translate from hVA to mPA. The virtual machine translation process is called a two dimensional page walk. Figure 2.2 illustrates the process; as shown, the 2D page walk can require up to 24 memory accesses.

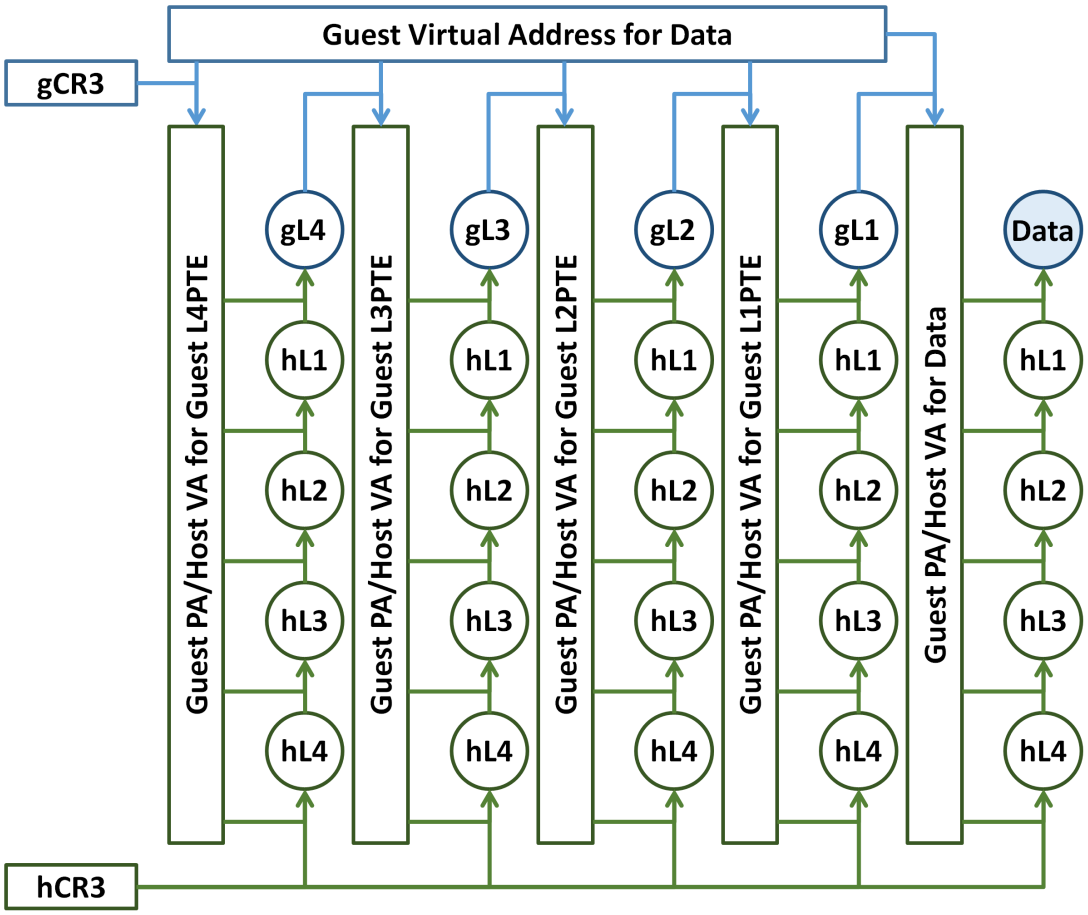

Figure 2.2: 2D Page Walk

2.1.2 Page Sizes

The base page size is the smallest unit of virtual-to-physical translation. In x86-64, the base page size is 4KB. Huge pages (also called superpages) simply refer to page sizes larger than the base page size. In x86-64, huge page sizes of 2MB and 1GB are currently supported. Two currently available software mechanisms in Linux are able to leverage huge pages: Transparent Huge Pages (THP) [18] and HugeTLBFS [19].

2.1.3 TLBs

Hardware must translate from virtual address to physical address on *every* memory access. Since page table walks are slow (they require potentially multiple memory accesses), processors use TLBs to cache recently used PTEs. On a TLB hit, the translation can be completed in one or a few cycles. On a TLB miss, a page walk is performed, often requiring tens or hundreds of cycles. The PTE produced by the page walk is typically installed in the TLB for future use. Current processors typically include separate instruction and data TLBs, with multiple levels of TLB.

Processors may include either separate set-associative L1 TLBs for each page size (e.g., Intel processors [37]), or a unified fully-associative L1 TLB for all page sizes (e.g., AMD, SPARC processors [63, 2]). L2 TLBs may be unified over multiple page sizes. Figure 2.3 illustrates the TLB hierarchy in a recent Intel Skylake processor [37].

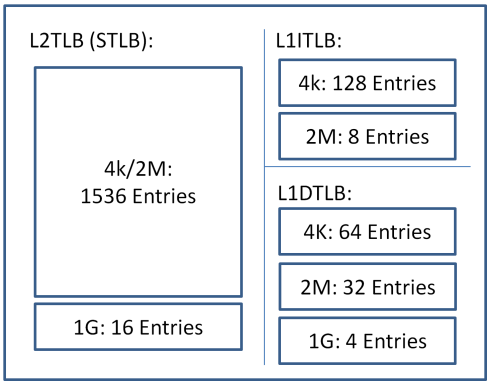

Figure 2.3: TLBs in Recent Intel Skylake Processor

It is worth noting that supporting multiple page sizes in a single set-associative TLB is non-trivial. The bits used to index the TLB are typically the least significant bits of the virtual page number. However, the size of the virtual page number depends on the page size, which is unknown while performing the TLB lookup.

In virtual machines, the TLBs may directly cache the gVA to mPA translation.

2.1.4 MMU Caches

Because a page walk would normally require four memory accesses (one for each level in the page table hierarchy), processors typically contain a memory management unit (MMU) cache which contains recently used PTEs from upper levels of the page table hierarchy. A hit in the MMU cache will reduce the number of memory accesses to fewer than four, the number of accesses depending on which level of page table hits in the cache.

In virtual machines, recently used PTEs from upper levels of the EPTs may also be saved in the MMU cache.

2.1.5 Larger Mapping Ranges

Several studies [62, 61, 8, 44, 45] have investigated larger mapping entities, orthogonal to huge pages. CoLT [62] and Clustered TLBs [61] leverage contiguous virtual pages that already map to consecutive physical frames due to the existing operation of the OS virtual memory system. This allows a single TLB entry to be used for up to 8 pages. Thus, a single extended 4KB TLB entry is able to reference up to 32KB (likewise, up to 16MB for a 2MB entry). Direct Segments [8] and Redundant Memory Mappings (RMM) [44, 45] utilize a segmentation-like translation mechanism that operates alongside the standard page table hierarchy and TLBs. Direct Segment only supports a single, large translation range, while RMM is significantly more versatile and can support many, arbitrarily-sized translation ranges. RMM will be discussed in further detail in Chapter 5.

2.2 OS Software for Virtual Memory

The operating system has several components responsible for virtual memory management. The OS maintains all processes' page tables and creates virtual-to-physical mappings upon a process request. I briefly describe the relevant components in the following paragraphs.

2.2.1 Buddy Memory Allocation

The buddy allocator tracks all free physical memory, keeping free lists of power-of-two sized blocks. Each free list is associated with a specific power-of-two size. When an allocation request for a new mapping occurs, the free list that matches the requested size is queried for a free block to use as the physical frame(s) for the necessary mapping(s). If there are no available blocks of the requested size, the free list containing blocks of the smallest size larger than the request is used. The larger-than-required free block is iteratively split in half to produce an appropriately sized free block. The two halves of any split block form a unique pair of buddy blocks (i.e., every block has a unique buddy block). When the split process finishes, the remaining left over free blocks are added to the appropriate free lists based on block size. When a process later deallocates and frees a block of physical memory, the allocator checks if its buddy block is also free. If so, the blocks merge, and the buddy block merge process repeats until the checked buddy block is not free. The block resulting from the merge operation is added to the appropriate free list. Thus, the buddy allocator splits and merges blocks of physical memory as appropriate during allocations and deallocations.

2.2.2 Demand Paging and Lazy Allocation

With demand paging, the operating system only performs page table setup upon receiving virtual-to-physical mapping requests. The OS marks PTEs initially as invalid since they do not yet actually point to physical frames. When a process first references a location on a newly mapped page, a page fault occurs, which notifies the operating system that a demand for the page exists. At this point, the page frame is appropriately selected and initialized. Although the operation of the buddy allocator and the prevalence of infrequent larger mapping requests easily lead to utilizing many contiguous page frames for contiguous virtual pages, this may not always be possible in cases of high allocation contention and interleaving of scattered demand requests.

2.2.3 Memory Compaction

The memory compaction daemon [20] is primarily responsible for reducing external fragmentation. With memory compaction, scattered blocks of used physical memory are migrated to adjacent locations to create larger, contiguous blocks of free memory. Memory compaction can also be explicitly invoked if sufficient contiguous memory cannot be found when the OS receives an allocation request.

2.2.4 Swapping

Frames in memory are typically backed by secondary storage. For example, when a process wants to read from a file, the operating system maps an area of the process's virtual address space to the file. This is called a file-backed mapping. The frame containing the in-memory version of the contents of the file is backed by the actual file on secondary storage. Thus, writes to the file which are carried out in memory must eventually be written to secondary storage. However, many mappings do not correspond to actual file contents.

For example, the stack and heap sections of a process are simply the process's in-memory working data. These mappings are called anonymous mappings. When physical memory demand across many applications exceeds the physical memory capacity, it is possible to swap currently lesser used anonymous mappings to secondary storage to free up physical frames for memory needed now. However, because I/O is orders of magnitude slower than memory, it is desirable to swap as infrequently as possible. Certainly, with continued increases in physical memory capacities, swapping continues to become less common.

2.3 Layout of a Program In Memory

Figure 2.4 shows the typical layout of a program in memory. Several of these regions (text, initialized data, uninitialized data (bss)) have statically known sizes and identical page protection bits throughout. In addition, data structures within dynamically allocated regions often have expected sizes known at runtime. As prior studies [44, 8] have identified, many applications only contain a small number of distinct, contiguous virtual regions, despite the large number of coarse-grained pages that are normally required to span the used virtual address space. In addition, the already standard operation of the operating system buddy allocator and memory compaction daemon often leads to consecutive virtual pages mapping to consecutive physical frames. These key observations lend motivation to my thesis. Ideally, only one virtual-to-physical mapping entity would be necessary to represent each truly unique virtual address region. My thesis provides the additional flexibility to allow an appropriately sized page to be chosen for each virtual region at memory allocation time.

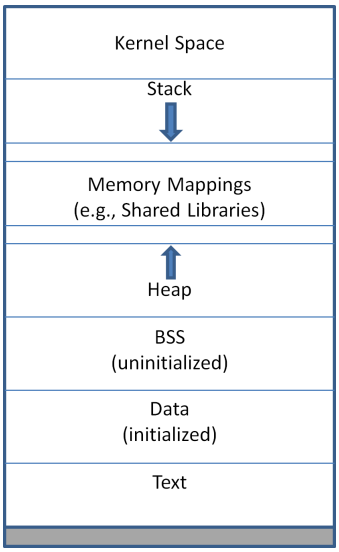

Figure 2.4: Typical Program Layout in Memory

Chapter 3

Tailored Page Sizes

Tailored Page Sizes (TPS) is a straightforward extension to current processor architectures. TPS reduces translation overhead, significantly lowers page walk memory references, and eliminates nearly all L1 TLB misses in TLB intensive workloads that will continue to scale with increasing physical memory capacity. TPS introduces support for tailored, fine-grained, alignment-constrained pages sized at any power-of-two larger or equal to the base page size. TPS includes changes to operating system software, the ISA, and the microarchitecture. However, it does not sacrifice backwards compatibility. The current paging model is still supported, with the additional flexibility of choosing from many newly available page sizes. When fragmentation is high, resulting in insufficient contiguity to well-utilize coarse-grained pages, TPS allows the OS to leverage what contiguity it can for performance with intermediate page sizes. The OS still has the option of solely choosing the original coarse-grained page sizes for whatever reasons may arise.

TPS leverages the natural levels of address space contiguity that occur due to the standard operation of the OS buddy allocators and the memory compaction daemon. Applications already perform runtime mapping system calls to large contiguous regions of their virtual address space, but current OS implementations split these mapping requests into however many smaller pages are required to compose the larger request. Utilizing these common properties, TPS is able to create the necessary virtual-to-physical mappings with

only a small number of appropriately sized fine-grained pages for many memory intensive applications. Each tailored page requires a single PTE cached in the TLB.

In the following sections, I discuss the architectural implementation details for TPS, the OS features needed to support TPS, and additional cross-layer considerations.

3.1 Architectural Considerations

3.1.1 Page Table and PTE Changes

To support TPS, the page table entry (PTE) structure and page walk process need to be updated. Figure 3.1 shows an overview of the typical x86-64 hierarchical page table. The current implementation supports three page sizes: 4KB, 2MB, and 1GB. For the 4KB page size, a page walk requires four physical memory accesses. If a larger page size is used, fewer accesses are required. For example, with a 2MB page, a bookkeeping bit in the 2nd level PTE identifies that this level is the final step of the page walk process. Similarly, for a 1GB page, a bit in the 3rd level PTE identifies that this level is the final step of the page walk process.

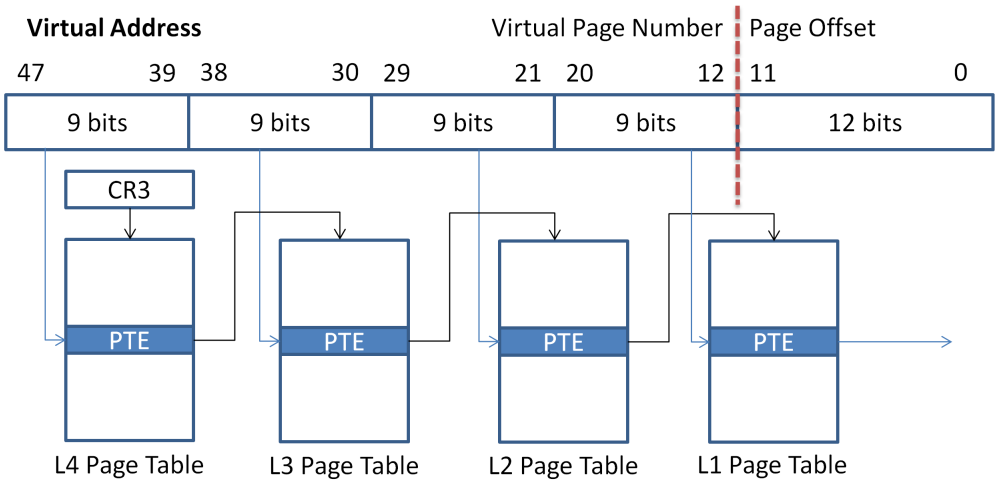

Figure 3.1: Four Level Hierarchical Page Table

To support pages at any power-of-two (larger than the base page) size, the PTE must include an additional field, as shown in Figure 3.2. There are nine page size options from 4KB up to (not including) 2MB. Thus, we use four reserved bits of the PTE to indicate what size page a given PTE is pointing to.

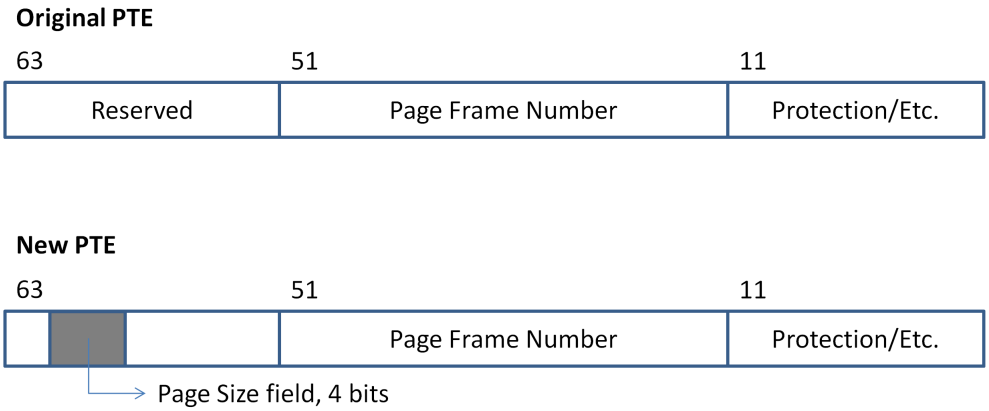

Figure 3.2: Original and Updated PTE

When performing the page walk, hardware has a challenge: the page size is not known until the PTE is read. Fine-grained page sizes may require one additional memory access for the page walk process. Figure 3.3 shows the details.

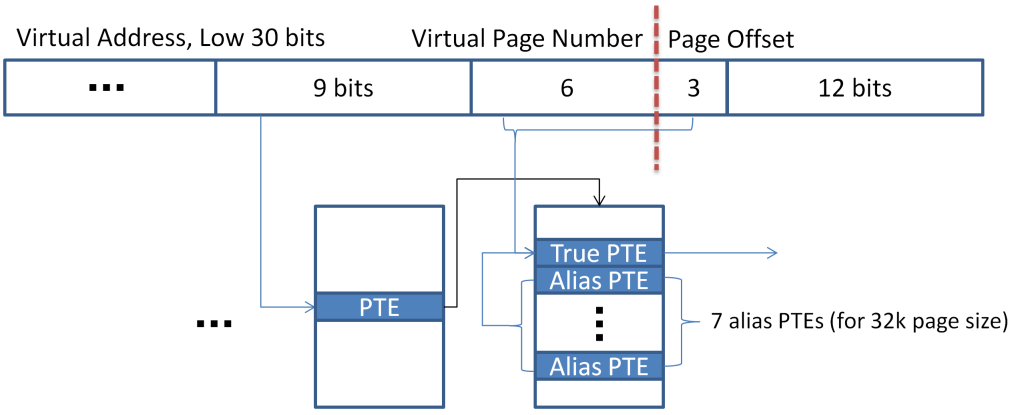

Figure 3.3: Extra Page Walk Access with Alias PTE

This example assumes a 32k page size, which implies the address has 15 bits of page offset. I call the nine bit subsection of a virtual address used to identify a specific PTE within the page table a *page table index*. Because this page table index is used to look up a 512 entry page table, a fine-grained page size may have multiple different page table index values that point to different PTEs, but actually represent the same page. Thus, when a fine grained page is created, all PTEs that could be pointed to by an address on that page are updated to indicate what the page size is. During the final access of the page walk process, the page size field in the PTE is examined. This field indicates how many bits of the nine bit subsection are not part of the virtual page number, but actually part of the page offset. Here, the additional memory access is performed with the bits of virtual address that are actually part of the page offset set to zero. This results in one PTE being the "true" PTE for the fine grained page, with the rest (called "alias PTEs") simply indicating one more access is necessary in the page walk. Note that the goal of TPS is to nearly eliminate TLB misses in most cases, so the one additional memory access required by only some page walks actually occurs rarely and is outweighed by the reduction in number of page walks (see Section 3.4.2). In addition, I expect the time spent to set up all the alias PTEs to be relatively inconsequential. With only coarse grained page support, these PTEs would need to be set up anyway as true PTEs for the numerous additional pages that would be created.

Reserved bits in the PTE are limited, so I also propose an alternative solution that only requires one reserved bit (called T in Figure 3.4). Note that larger pages have fewer page frame number (PFN) bits. For example, assume 40 bits of physical address. A 4KB page has 12 bits of page offset and 28 PFN bits, while an 8KB page has 13 bits of page offset and 27 PFN bits. The one reserved bit (T) specifies whether the PTE corresponds to a standard coarse-grained page (e.g., 4KB), or a tailored intermediate page (e.g., >4KB). If the page is tailored, the PTE must now have at least 1 bit of PFN that is thus unused

(e.g., bit s_0 in Figure 3.4). This PTE bit (that would otherwise be part of a PFN) specifies whether the page size is the smallest tailored size (e.g., 8KB), or larger (e.g., >8KB). If the page is larger, then the PTE must have yet another unused PFN bit (e.g., bit s_1), and this process can repeat. This can be easily implemented in hardware with a priority encoder to identify the page size. Figure 3.4 and Table 3.1 illustrate the process.

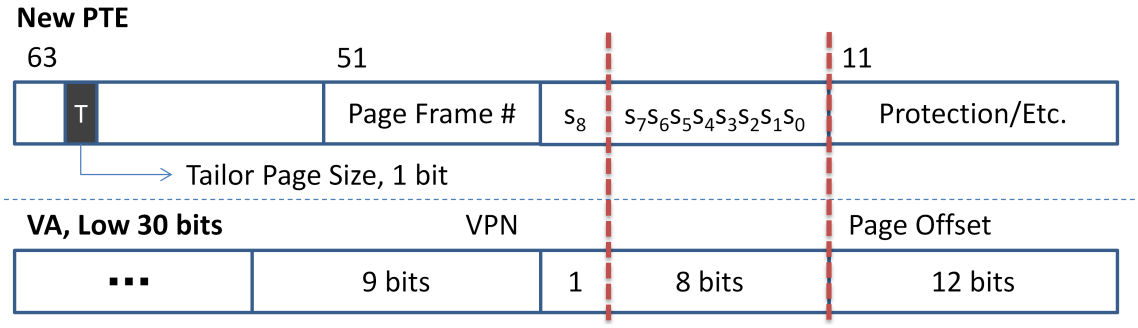

Figure 3.4: Identifying Page Size with Only One PTE Bit

s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0	T	**Page Size**
X	X	X	X	X	X	X	X	X	0	4k
X	X	X	X	X	X	X	X	0	1	8k
X	X	X	X	X	X	X	0	1	1	16k
X	X	X	X	X	X	0	1	1	1	32k
X	X	X	X	X	0	1	1	1	1	64k
X	X	X	X	0	1	1	1	1	1	128k
X	X	X	0	1	1	1	1	1	1	256k
X	X	0	1	1	1	1	1	1	1	512k
X	0	1	1	1	1	1	1	1	1	1m

Table 3.1: Page Size Logic

3.1.2 TLB Design

The TLB design must be augmented to support TPS. A recent Intel Skylake Processor [37] includes two levels of TLB for data accesses. The L1 TLB is split into three

parts for the three supported page sizes. It contains 64 entries for the 4KB page size, 32 entries for the 2MB page size, and 4 entries for the 1GB page size. The L2 TLB contains 16 entries for the 1GB page size, and 1536 entries for the 4KB/2MB page sizes. To support TPS, I modify the L1 TLB to contain a 32 entry, fully-associative (as in other commercial designs [2, 63]) TPS TLB. The TPS TLB takes the place of the existing 32 entry and 4 entry larger page size L1 TLBs. The TPS TLB supports any page size. I still retain the 64 entry 4KB L1 TLB. Alternative skewed-associative [70, 60] TLB designs are possible.

In the newly added L1 TPS TLB, I add a *page mask* field to each TLB entry, as shown in Figure 3.5. Changes to existing hardware are contained in the dashed box. The page mask field is populated when the TLB is filled. The TLB is searched by the Virtual Page Number (VPN) of the memory access. Normally, the VPN is compared to the VPN tag stored within the TLB to identify a hit. In the case of our any-page size TLB, the incoming VPN is first masked with the entry's mask field, then compared to the VPN tag to identify a hit. This adds a single gate delay to the TLB lookup, which is unlikely to impact the observed latency of the L1 TLB lookup. This is effectively a ternary CAM approach.

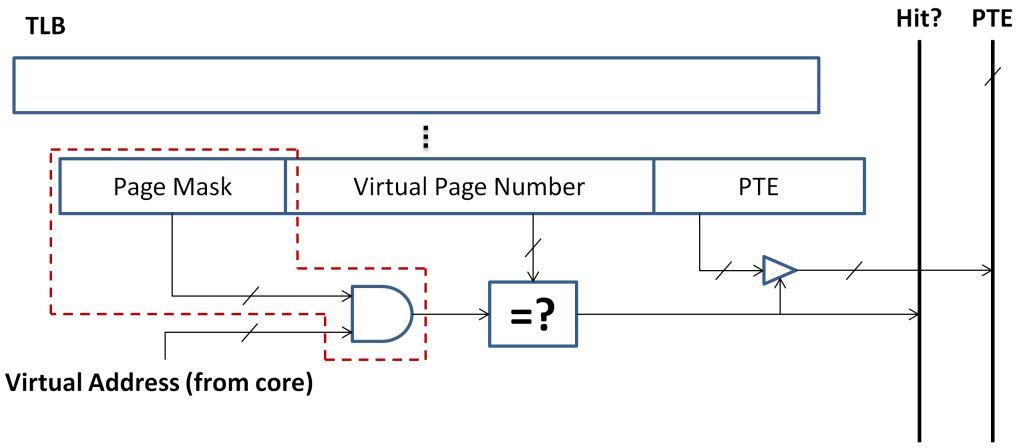

Figure 3.5: TLB Hardware

3.1.2.1 L2 TLB

Existing L2 TLBs share entries between the 4k and 2M page sizes. It is publicly unknown exactly what technique processors use to share these entries across two page sizes in a set-associative design, but the possible approaches fall into two general categories: sequential or parallel lookup [12]. Hash-rehashing enables sequential TLB lookups for each page size, but if the wrong size is looked up first, there will be additional latency to look up the second size. Parallel lookup is possible via various approaches (e.g., dual porting the entire TLB; duplicate/dual-ported tag store with serial tag then data lookup). Identical L2 TLB access times for both page sizes [39] suggests some kind of parallel lookup is performed.

Using only sequential or parallel lookup for all the different tailored page size options is costly. Sequential lookup would require tens of serialized lookups to check all page sizes, and parallel lookup would require significant area increase for the additional ports for each page size. A better solution is necessary to prevent these costs.

TPS uses a combination of four simple approaches to minimize L2 TLB cost. First, search multiple consecutive page sizes in a single lookup (e.g., 4k, 8k, 16k). Second, distribute page sizes accross parallel lookup ports. Third, sequentially lookup across page sizes. Fourth, tile page sizes accross TLB entries by assigning some ways to specific page sizes. The most common page sizes should be mappable to as many TLB entries as possible and looked up in the first access.

For a 1536 entry, dual ported, 12-way L2 TLB, the simple approach distributes page sizes accross three dimensions: access number (i.e., using hash-rehash), ports, and ways. For access 1, port 1, all 12 ways: sizes 4k, 8k, 16k are checked. For access 1, port 2, all 12 ways: sizes 2M, 4M, 8M are checked. See Table 3.2 for the detailed breakdown. Five bits

of extra tag per TLB entry for page size are sufficient to implement this. Sizes larger than 8G could be shared inside the existing fully-associative 16-way 1GB L2 TLB.

Even more complex tiling leveraging the dimension of TLB sets could result in better L2 TLB efficiency. Intelligent tiling should leverage the fact that larger pages require fewer entries.

(Access #,	Port #,	Way #s)	Page Sizes
(1,	1,	1-12)	4k, 8k, 16k
(1,	2,	1-12)	2M, 4M, 8M
(2,	1,	1-6)	32k, 64k
(2,	1,	7-12)	128k, 256k
(2,	2,	1-6)	512k, 1M
(2,	2,	7-12)	16M, 32M
(3,	1,	1-6)	64M, 128M
(3,	1,	7-12)	256M, 512M
(3,	2,	1-6)	1G, 2G
(3,	2,	7-12)	4G, 8G

Table 3.2: L2 TLB Tiling

3.1.2.2 TLB Reach

The changes to the TLB hierarchy improve TLB reach even without utilizing Tailored Page Sizes. Assuming only existing coarse-grained page sizes are used, Table 3.3 breaks down TLB reach by page size and TLB level for the baseline and TPS TLB configurations. TLB reach significantly improves, largely due to the significant increase in the number of TLB entries that can be used by the 1 GB page size. However, using the large 1 GB size for every allocation to achieve maximal reach increases the potential loss of memory to internal fragmentation.

TLB Level	Page Size	TLB Configuration			
		Baseline		TPS	
L1	4 kB	256	kB	256	kB
	2 MB	64	MB	64	MB
	1 GB	4	GB	32	GB
	All Entries	4.06	GB	32	GB
L2	4 kB	6	MB	6	MB
	2 MB	3	GB	3	GB
	1 GB	16	GB	1.5	TB
	All Entries	19	GB	1.5	TB

Table 3.3: TLB Reach When Only Using Coarse-Grained Page Sizes for Baseline and TPS TLB Configurations

The TLB reach when Tailored Page Sizes are enabled depends on what the maximum supported page size is. Assuming a maximum page size of 16 GB, TPS has an L1 TLB reach of 512 GB and an L2 TLB reach of just under 8 TB.

3.2 Operating System Considerations

3.2.1 Paging and Buddy Allocation

While the standard approach in demand paging does allow intermediate contiguity of mapped physical frames, changes are necessary to extract the full potential of Tailored Page Sizes. Utilizing an eager paging strategy as in [44] is possible. Rather than lazily allocating coarse-grained pages on demand when first accessed, we identify the appropriate tailored page size and eagerly allocate the few fine-grained pages (which use appropriately sized frames of physical memory provided by the buddy allocator) at allocation request time (e.g., when the process performs an mmap system call).

However, there are some drawbacks to changing the OS's paging strategy. Application start-up time and allocation latency may be adversely affected if the application

must wait for entire large pages to be initialized (which is also a problem using standard large page sizes). Additionally, the larger the page size, the more costly the swapping. But, swapping is becoming less common: servers running big memory workloads often run with swapping disabled, preferring to keep entire working sets in memory to minimize latency [8, 48]. Apple iOS also does not swap to secondary storage [3]. The continuing increase in physical memory capacity significantly reduces the frequency of swapping [8].

To further improve the robustness of TPS in the cases where these concerns present significant problems, I utilize an alternative to both demand and eager paging: *demand paging with frame reservation*. My approach is similar to the reservation based paging strategy used in FreeBSD [55, 15] and previously proposed in [58, 75]. Reserved frames are neither free nor in use; they can transition to either state depending on system demands.

When a large size allocation request occurs, the operating system still identifies the desired optimal fine grained page size N, but does not initially allocate the entire region as in standard demand paging. Instead, the buddy allocator is queried for a free memory block of size N, which is then removed from the allocator free list and placed into a *paging reservation table* that also saves the requested range of virtual addresses. This free memory block of size N is reserved for virtual addresses within the range.

When the first demand request (memory access) occurs to a location within that range, only the coarse grained page containing the demand request is allocated (as in standard demand paging). The appropriate frame is chosen from the block of size N previously saved into the paging reservation table, rather than the buddy allocator free list. For a subsequent demand request to a yet-to-be-mapped part of the virtual address range, the already mapped page is grown (also called “page promotion”, or “upgrading the page size”) to in-

clude the location of the new request. The physical memory locations are identified from the paging reservation table.

Upgrading the page size simply requires updating the appropriate PTEs for the newly-mapped larger page. While the newly mapped memory must be appropriately initialized, no changes to or migration of the previously mapped frame is necessary. Figure 3.6 shows an initially empty reservation incrementally receiving demand requests that gradually create larger pages, until the reservation is filled and only one tailored page is ultimately needed.

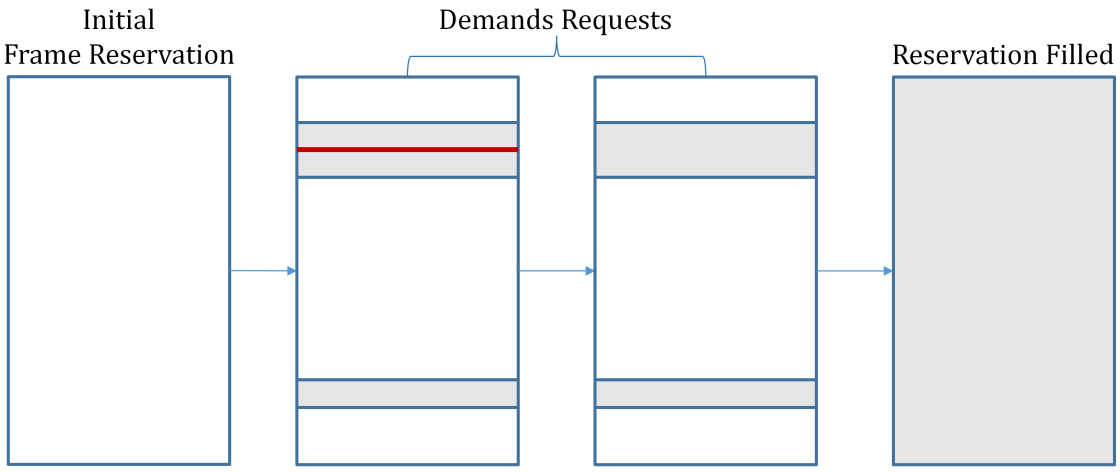

Figure 3.6: Incrementally Filling Out a Page Reservation

TPS's support for page sizes at every power-of-two allows frame reservations to be incrementally filled with growing page sizes by the OS as demand requests within the reserved region arrive. Unlike the FreeBSD's approach, TPS can adjust page promotion aggressiveness based on a utilization threshold. To prevent memory footprint bloat, TPS can be configured to only upgrade to larger page sizes when 100% of the larger page's constituent pages are utilized. Conversely, for better TLB performance, TPS could also be configured to upgrade to larger page sizes when lower percentages (e.g., 50%) of the

constituent pages are utilized. The promotion threshold can be adjusted between the two extremes to balance the tradeoff based on the machine's memory load.

These straightforward changes to the paging algorithm and allocator allow the OS to map an application's utilized virtual address space with a minimal number of appropriately sized pages.

3.2.2 Fragmentation

A primary general drawback of supporting more than one page size is fragmentation. When fragmentation does become a problem, TPS always has the simple fall back of only allocating from the originally supported coarse grained page sizes. In addition, the OS can request memory compaction when fragmentation is high to present more opportunities for fine-grained tailored allocations and merges, as with standard coarse-grained allocations. External fragmentation occurs when free memory blocks and allocated memory blocks are interspersed. This can prevent a large contiguous allocation even though total free memory exceeds the allocation size. Internal fragmentation occurs when part of an allocated memory block is unused.

3.2.2.1 External Fragmentation

To minimize external fragmentation, TPS conservatively only upgrades page reservations when utilization is near 100%, as described in Section 3.2.1. As external fragmentation increases, the OS will be unable to create desired page sizes and reservations due to the lack of memory contiguity, and be forced to create smaller pages. Under heavy external fragmentation, coarse-grained large pages cannot be created; however, it is possible that whatever minimal memory contiguity is available can be leveraged by TPS to create intermediate page sizes.

3.2.2.2 Internal Fragmentation

With larger pages, the potential for more waste due to internal fragmentation is increased. The most conservative policy is to completely disallow any extra loss due to internal fragmentation (as compared to exclusive use of the smallest page size) by creating reservations out of the fewest number of pages that exactly spans the reservation (e.g., an aligned 28kB request results in 16kB+8kB+4kB).

On the aggressive side, TPS can choose the smallest size page still larger than the requested memory allocation. Because only power-of-two page sizes are supported, this could lead to approximately 50% waste in some cases. For example, an allocation request of 2052 KB would result in a 4 MB page reservation being created. When fragmentation is high, TPS can throttle towards more conservative page size reservations. For example, if the ideal page size allocation results in greater than or equal to 75% utilization, then the OS will create one page at the ideal size. Otherwise, the OS will create two pages: the first page will be half the ideal size, and the second page will be the ideal page size for the remaining portion of the allocation, see Figure 3.7. With an allocation request size of 23 KB, the aggressive policy creates a 32 KB page, resulting in at least 50% utilization. The more conservative policy creates two pages, but guarantees at least 75% utilization. This choice presents a tradeoff between magnitude of internal fragmentation, and the number of TLB entries required to translate a single logical region.

Figure 3.7: Fragmentation Tradeoff

3.2.3 Memory Compaction and Page Merging

TPS does not require any changes to the standard memory compaction daemon operation to improve performance. Typical operation of the memory compaction daemon is primarily concerned with packing used blocks of physical memory to improve the availability of larger-sized, contiguous, free blocks of memory. Long-running large footprint benchmarks already benefit from upfront compaction at application launch to create coarse-grained pages; TPS maximizes the benefit from this sort of compaction.

A possible future optimization might be to allow the memory compaction daemon to be aware of physical frames that could be potentially merged into a single larger frame that would require only one PTE for translation. A page merge is possible when two appropriately aligned and adjacent frames map contiguous virtual address space with identical permissions. Multiple allocations to contiguous virtual addresses may have been unable to use contiguous frames due to other intervening allocations. Whenever memory compaction is performed, the daemon could migrate frames taking into account potential merges. When the PTEs pointing to migrated pages are updated, the OS performs a page merge by updating the relevant PTEs (and invaliding appropriate TLB entries) if the frames were successfully setup for a merge.

3.3 Other Considerations

3.3.1 PTE Accessed and Dirty Bits

The processor is required to update the Accessed and Dirty (A/D) bits for a given PTE when loads read from (stores write to) a page with the bit cleared. The TLB also caches these bits to identify whether the additional store to update the PTE will actually be required. One concern with larger pages in general is that keeping track of accessed and

dirty data at a larger granularity may incur additional overheads during swapping and writing back dirty pages to secondary storage. Current large page sizes already have to deal with this problem; there is no additional overhead introduced by supporting more intermediate page sizes. However, as with large pages, when fragmentation is high, swapping frequent, or I/O pressure high due to cleaning dirty pages, the OS has the option of splitting larger pages into smaller pages to reduce the associated costs.

As an alternative, recall that intermediate tailored page sizes will have multiple alias PTEs which are simply used to point to the true PTE. The remaining bits in the alias PTEs are thus unused and could be collected into a bit vector representing the referenced/modified state of a tailored page's constituent coarse-grained pages. The vector can be cached with the TLBs and does not actually need to be loaded on a page walk because the A/D bits exhibit sticky behavior. The first read/write will update the in-memory A/D bit to guarantee it is set, and update the cached TLB bit to prevent extraneous updates. Note that this bit vector need not be strictly tied to the TLB lookup; the bit vector's lookup and update operations can proceed after the standard TLB lookup in parallel with the subsequent memory access pipeline stages. A tailored page can have up to 256 constituent coarse-grained pages. Since tracking up to 512 bits may be too costly both in terms of TLB area and additional memory accesses required, we can impose an upper bound on the bit vector length. For example, a 16 bit limit would significantly reduce costs while still allowing for fine-grained tracking. Each bit's tracking granularity would be a function of the page size. A bit in the PTE can specify whether to enable or disable this fine-grained metadata tracking. The bit vector updates use the same mechanism already used by the existing modify bit update operation and do not block forward progress.

When two pages of size N merge into a page of size $2{\cdot}N$, the bit vectors need to be updated. Figure 3.8 shows the simple logic to generate the new vector. The cached vectors in the TLB for the original smaller pages are still valid, but may result in extraneous stores for A/D bits that are already set.

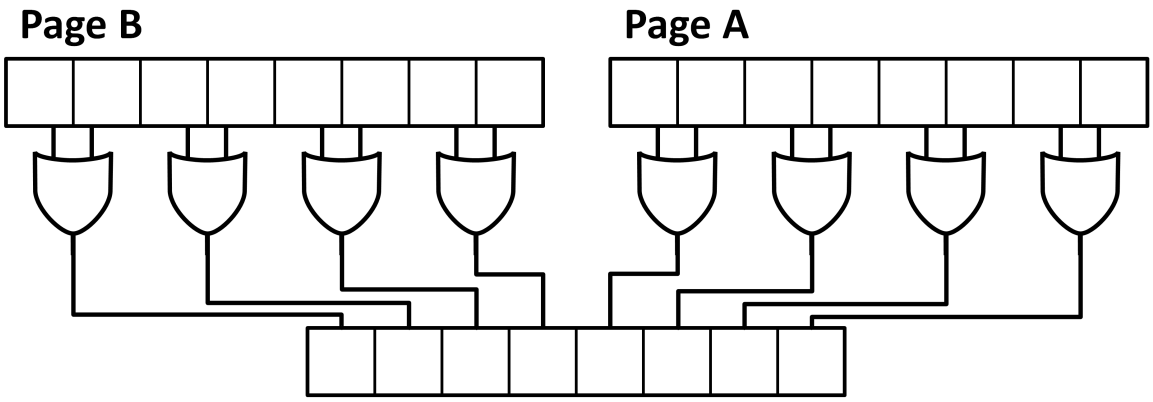

Figure 3.8: Merging Access/Dirty Metadata Vectors

3.3.2 TLB Shootdowns

In x86-64, the INVLPG instruction is used to invalidate any out-of-date PTEs that may be stored in the processor TLBs. No changes are needed to the operation of this instruction. As in standard operation, appropriate shootdowns remain necessary during memory compaction. When a page results from merging adjacent pages, the PTE that may have been present in a TLB for the smaller page size will still be correct for its portion of the larger page. For optimal TLB entry replacement, the ideal policy is to only update LRU information for the largest page size which returns a hit. This is unnecessary, however, because as pages grow, the likelihood of extraneous smaller page TLB entries being aged out increases. Thus, no new shootdowns are needed when performing page merges.

3.3.3 Copy on Write

Copy on Write is an OS technique that enables multiple virtual pages that contain identical data to point to the same physical frame by maintaining the PTE in read-only state. When a page is written to, a page fault occurs, and the OS copies the frame and updates the mapping. With larger pages, opportunities to use copy-on-write will be reduced, simply because there is lower likelihood such large regions of memory are identical. If there is substantial desire to share a particular small page, the OS can simply prioritize using a smaller page for such a page. If a larger page is read shared, then upon a write, the OS has multiple options. The OS could only copy the part of the larger page that was written to as a smaller page, and create multiple larger page mappings still sharing the parts of the page that were not written to. This saves copy time and reduces memory utilization. Alternatively, the OS could copy the entire large range, which is more expensive in terms of copy time and memory utilization, but reduces TLB pressure.

3.4 Evaluation

3.4.1 Methodology

To evaluate the performance impact of TPS, I performed a two-step evaluation in simulation. Implementing the proposed OS changes in an existing OS requires a physical processor with the proposed ISA support. Also, it is preferable to run workloads from start to finish in order to fully understand the TLB behavior and translation overheads of memory-intensive applications. However, doing this on a cycle-based or full system simulator is infeasible; months of simulation time would be required.

To study the impact of TPS, I primarily evaluate how TLB hit rates are affected. I constructed a PIN-based OS-allocator and virtual memory simulator that traces memory

management system calls and all memory accesses. The simulator models the relevant parts of the microarchitecture and operating system. I modeled a realistic TLB hierarchy with MMU caches, and identified the number of accesses, hits, and misses to each level of the hierarchy. Additionally, I modeled the required OS changes including those to the allocator and application page tables. The PIN-based virtual memory simulator does not include the instruction/data cache hierarchy.

To demonstrate the importance of hitting in the L1 TLB, I used ZSim [66], a cycle-based simulator of an x86 superscalar out-of-order processor. The simulator has been strongly correlated to existing Intel processors and faithfully models core microarchitectural details and the memory hierarchy. I added TLBs to the simulator to model the impact of TLB misses. Evaluation is performed on a single-core system. Table 3.4 lists the baseline processor configuration.

Core	4-Wide Issue, 256 Entry ROB, 3.2 GHz Clock Rate
L1$	32KB I$, 32KB D$, 64 Byte Cache Lines, 4-Cycle Latency, 8-Way Set Assoc.
LLC	2MB, 16-way Set Associative, 64 Byte Cache Lines, 10-Cycle Latency
TLBs	128 4k + 8 2M L1ITLB, 1-Cycle Latency 64 4k + 32 2M + 4 1G L1DTLB, 1-Cycle Latency 1536 4k/2M + 16 1G STLB, 14-Cycle Latency
MMU	L2 Page Table Cache: 64-Entry, 4-Way Set Associative, 2-Cycle Latency L3 Page Table Cache: 16-Entry, 4-Way Set Associative, 2-Cycle Latency L4 Page Table Cache: 4-Entry, Fully Associative, 2-Cycle Latency

Table 3.4: Simulated Processor Configuration

The benchmarks I evaluated are the SPEC17 suite, Graph 500, GUPS, XSBench, and DBx1000. Since the SPEC17 suite is not fully representative of modern TLB intensive workloads, I profiled all the benchmarks as shown in Figure 3.9 to determine each benchmark's TLB pressure, as measured in TLB misses per thousand instructions (MPKI). For

evaluation, I chose the TLB intensive SPEC17 benchmarks, as measured by an MPKI of greater than five. Details of the chosen benchmarks are shown in Table 3.5.

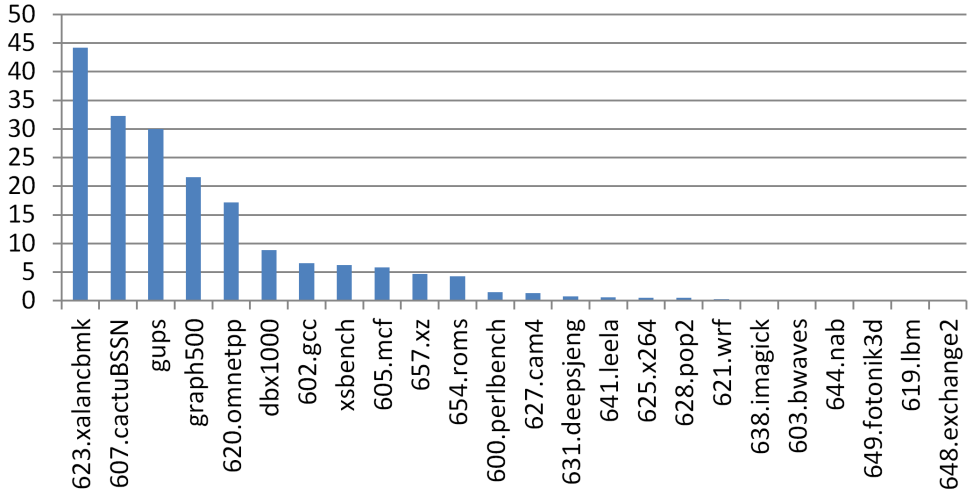

Figure 3.9: L1 DTLB MPKI

Benchmark	**Description**
gcc	Spec17, C compiler
cactuBSSN	Spec17, physics equation solver
omnetpp	Spec17, ethernet discrete event simulation
xalancbmk	Spec17, XML processor
mcf	Spec17, scheduling optimization problem
Graph500	graph construction/BFS/SSSP
GUPS	random memory access
XSBench	neutron particle transport simulation
DBx1000	OLTP in-memory database

Table 3.5: Benchmarks

Real system evaluation using performance counters was carried out on a machine with a recent Intel Kaby Lake processor and 64 GB of DDR4 memory.

3.4.2 Results

3.4.2.1 Translation Overheads

There are two primary sources of translation latency in current processors:

1. L2 TLB misses that result in page walks.
2. L1 TLB misses that still hit in the L2 TLB.

To quantify the real cost of page walks, I collected performance counter data on a real machine. Figure 3.10 shows the percentage of total application execution time the processor spends on page walks, with Transparent Huge Pages active. Three cases are shown:

1. Native execution with no interference.
2. Native execution with a simultaneous multi-threading (SMT) hardware thread competing for TLB resources.
3. Virtualized execution with two dimensional page walks.

Figure 3.11 shows the performance improvement of a perfect L1 TLB over a perfect L2 TLB baseline, configured to represent the existing Transparent Huge Pages approach. This study was performed with cycle-based simulation modeling out-of-order effects on ZSim. The out-of-order window can often hide many L1 TLB misses by overlapping this latency with other useful work. But, when memory accesses are on the critical path of execution (e.g., linked data structure traversal), even frequent L1 TLB misses can cause an appreciable performance penalty as shown.

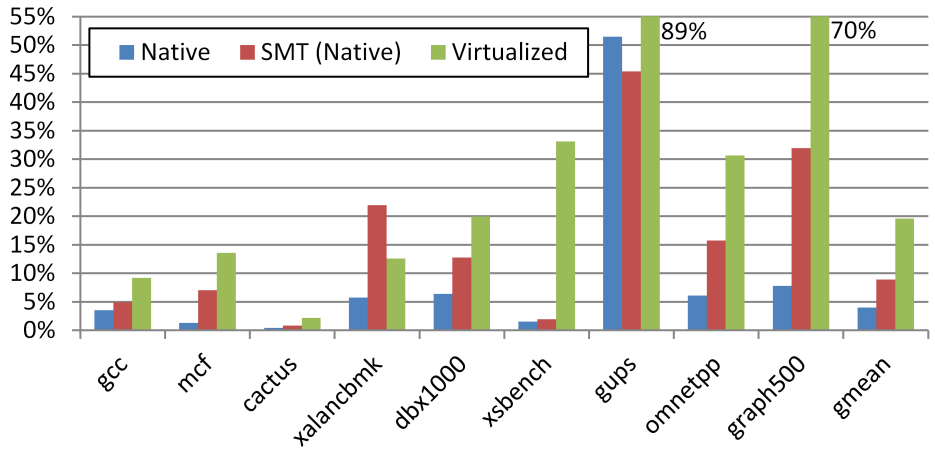

Figure 3.10: Page Walk Overhead: Percent of Execution Time Spent Page Walking

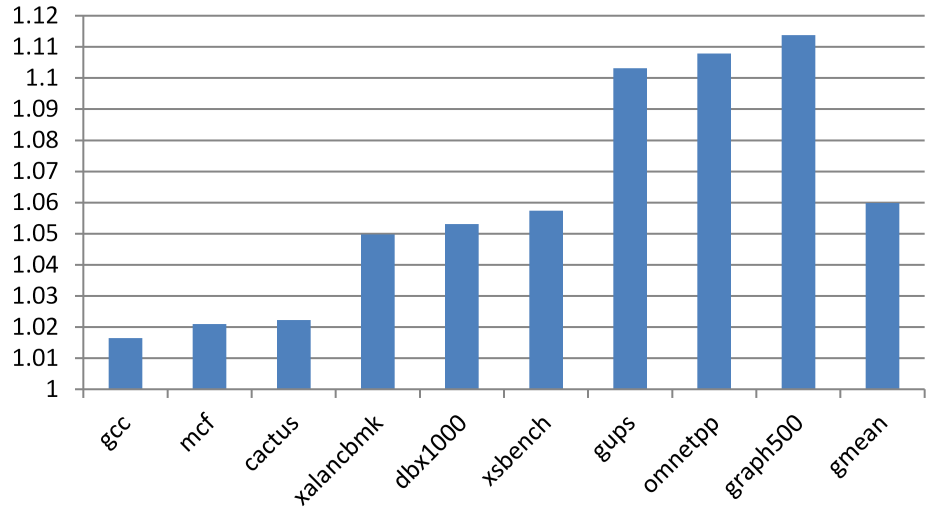

Figure 3.11: L1TLB Miss Overhead: Speedup of Perfect L1 TLB over Perfect L2 TLB Baseline

3.4.2.2 Memory Utilization

Prior OS work [49] observed that Linux Transparent Huge Pages can result in memory footprint increases of up to 70% for some benchmarks. In my PIN-based simulator, I simulated running with only 2MB pages to see the maximum memory size impact of using only 2MB pages compared to using only 4KB pages on the benchmarks. Results in Figure 3.12 show only modest increase in memory utilization for the benchmarks, but it is important to note that mixing in the 1GB page size increases the potential memory loss due to internal fragmentation. For all remaining experiments, I ran TPS under the frame reservation strategy, requiring 100% utilization of smaller page sizes before merging into the next larger tailored page size. This approach guarantees identical memory usage to using only the base page size of 4KB, but loses some opportunity to reduce TLB misses.

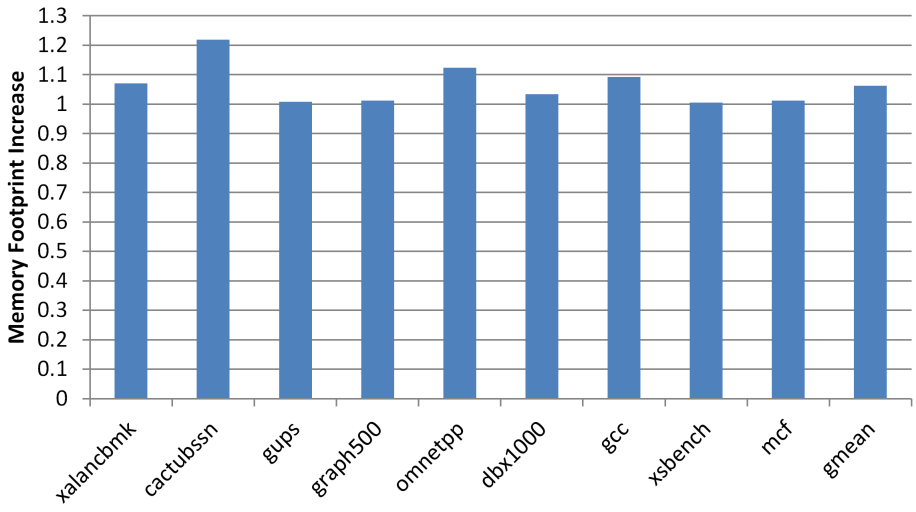

Figure 3.12: Increase in Memory Utilization with Exclusive 2MB Pages

3.4.2.3 Reduction in TLB Misses

Figure 3.13 shows the PIN-based simulation results. No migration or compaction is performed in this study, but initial memory state is assumed to be lightly-loaded. I report the percentage of L1 DTLB misses eliminated for each benchmark with TPS enabled compared to the baseline of reservation-based Transparent Huge Pages (THP). I also implemented and evaluated the impact of CoLT [62] and RMM [44] on the L1 TLB hit rate.

As shown, TPS eliminates 98.0% of L1 TLB misses on average, while CoLT only eliminates approximately 36.6% of L1 DTLB misses on the benchmarks. RMM eliminates no L1 DTLB misses. This is because RMM introduces a Range TLB at the L2 level of the hierarchy. CoLT has minimal impact for the benchmark GUPS because of its random access behavior. Increasing the reach of each TLB entry by a small factor does little to mitigate a random memory access pattern to gigabytes of physical memory. In contrast, tailoring a few very large pages to the size of this memory region significantly reduces TLB misses.

Figure 3.14 shows the reduction in the number of page walk memory references. TPS and RMM have near identical best performance in terms of maximal reduction in page walk memory references. RMM generally slightly outperforms TPS primarily because RMM has no size or alignment restrictions on large regions. While eager paging is best for page walk reduction, this policy can have unacceptable impact on allocation latency in some scenarios. TPS slightly outperforms RMM on the gcc benchmark because of the limited number of available Range TLB entries.

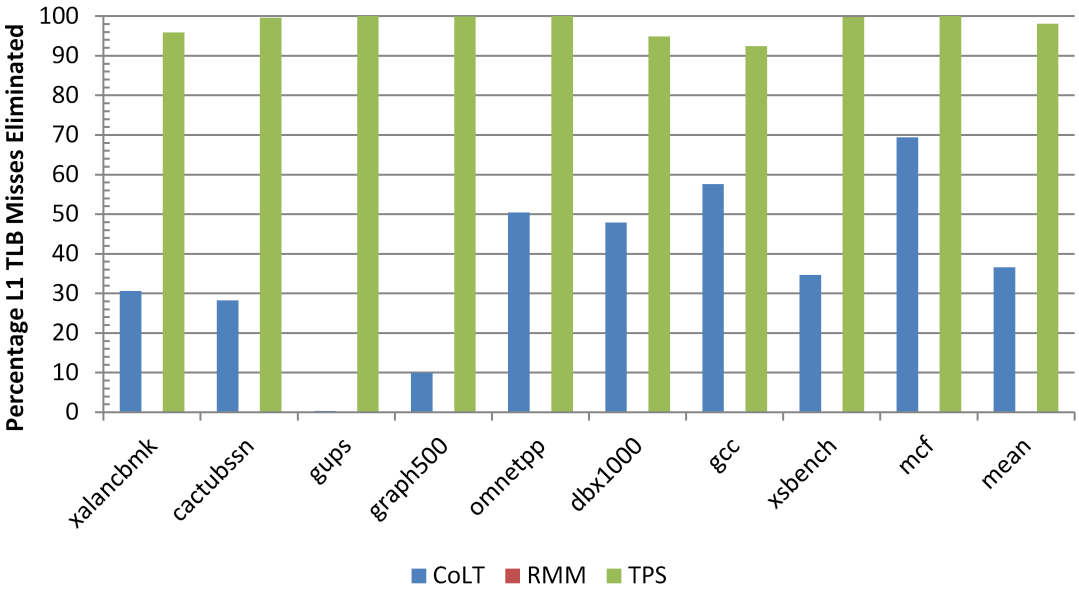

Figure 3.13: L1 DTLB Misses Eliminated

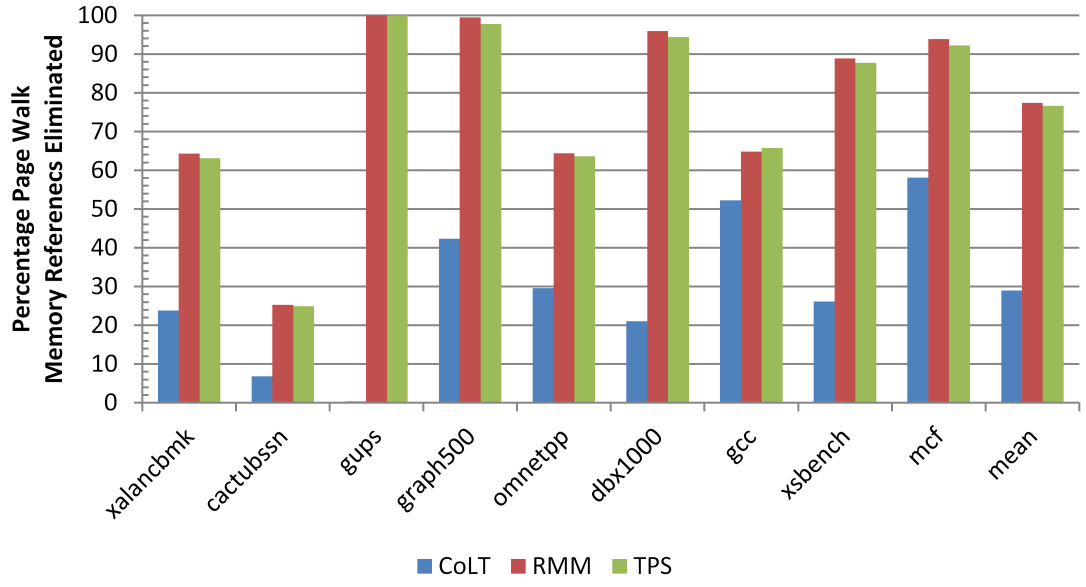

Figure 3.14: Page Walk Memory References Eliminated

Table 3.6 shows the hit rate for each level of MMU cache (i.e., there are three separate MMU caches, one for each of the L2, L3, and L4 page tables). The L2 page table cache (L2PT$) only increments access/hit/miss counters for L2TLB misses; similarly, the L3PT$ only increments access/hit/miss counters for L2PT$ misses (and same for the L4PT$). Hit rates are reported for two configurations: 1) the baseline THP, and 2) TPS.

	Baseline (THP)			**TPS**		
Benchmark	**L2PT$**	**L3PT$**	**L4PT$**	**L2PT$**	**L3PT$**	**L4PT$**
xalancbmk	98.6%	99.9%	0.0%	99.7%	98.9%	0.0%
cactubssn	72.4%	100.0%	72.7%	99.7%	99.8%	72.7%
gups	1.2%	100.0%	69.2%	99.8%	99.7%	69.2%
graph500	80.3%	100.0%	33.3%	99.8%	99.5%	33.3%
omnetpp	99.9%	98.3%	33.3%	99.7%	98.3%	33.3%
dbx1000	3.8%	100.0%	78.6%	99.8%	99.7%	78.6%
gcc	58.1%	100.0%	75.0%	97.5%	100.0%	75.0%
xsbench	35.8%	100.0%	60.0%	99.8%	99.7%	60.0%
mcf	39.4%	100.0%	77.8%	99.8%	99.7%	77.8%
Mean	54.4%	99.8%	55.6%	99.5%	99.5%	55.6%

Table 3.6: MMU Cache Hit Rates (TPS)

In all cases, the actual total number of page walks is significantly reduced with TPS. The hit rate of the L2PT$ improves to over 99% on average with TPS because it is consulted much less frequently; and, when it is consulted, there is less contention for entries.

3.4.2.4 Performance Estimation

To approximate performance impact, I break down total execution time, T, into 3 parts:

$$T = T_{IDEAL} + T_{L1DTLBM} + T_{PW}$$

where

- T_{IDEAL} is the ideal execution time of the benchmark assuming no page walks or TLB misses.
- $T_{L1DTLBM}$ is the execution time lost due to L1TLB misses that hit in the L2TLB (did not cause page walks).
- T_{PW} is the execution time lost due to page walks.

For the THP baseline case, I measured $T_{L1DTLBM}$ via ZSim simulation, as shown in Section 3.4.2.1.

The performance counter measurements from Section 3.4.2.1 cannot be directly used to calculate T_{PW}. The performance counters measure cycles where a page walker is active; this means that it is possible for an application to be making forward progress due to the out-of-order window even while the page walker is active. Thus, eliminating page walker cycles will not always translate to a one-to-one reduction in execution time. To estimate how much a reduction in page walker cycles will actually be realized as savings in execution time, I collected real machine performance counter information at two configurations, measuring two values at each configuration:

1. Transparent Huge Pages disabled (only 4k pages).
   (a) Total Execution Cycles: TC_{THP_d}
   (b) Page Walker Cycles: PWC_{THP_d}
2. Transparent Huge Pages enabled.
   (a) Total Execution Cycles: TC_{THP_e}
   (b) Page Walker Cycles: PWC_{THP_e}

From these two points, I calculated how a reduction in PWC translated to a reduction in TC going from THP disabled to THP enabled. Assuming the trend holds, I can then estimate how much further reductions in PWC will translate to total execution time savings. Figure 3.15 shows the percentage of PWC savings that directly translates to total execution time savings for each benchmark.

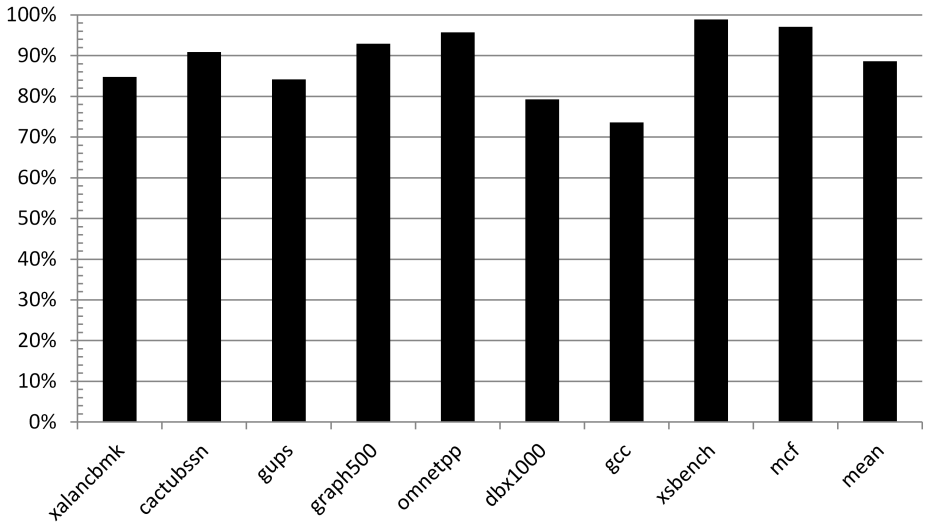

Figure 3.15: Savable Page Walker Cycles

I use this performance counter information to calculate T and T_{PW}. Using the equation above, I can solve for T_{IDEAL}.

To determine T with TPS active, I scale T_{PW} by the ratio of page walk memory references eliminated (as determined via simulation), and I scale $T_{L1DTLBM}$ by the ratio of L1 DTLB misses eliminated. I perform the same calculations for CoLT and RMM. Figure 3.16 shows the speedup results for native execution, running on a core alone.

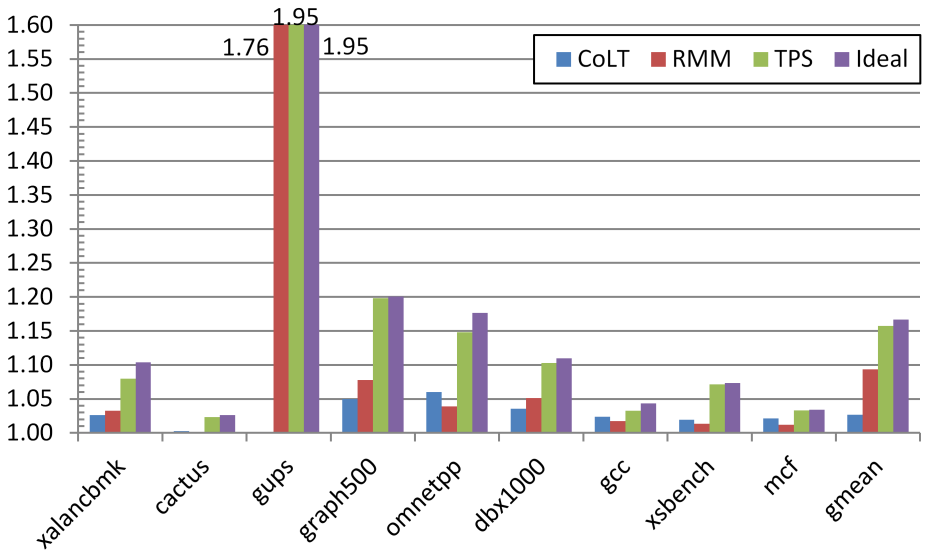

Figure 3.16: Speedup - Native (no SMT)

TPS is able to achieve an average performance improvement of 15.7%, as compared to 9.4% for RMM and 2.7% for CoLT, which is 99.2% of the maximal ideal savings (i.e., eliminating 100% of TLB misses) for TPS.

Figure 3.17 shows the speedup results for native execution, running with a hardware SMT thread competing for core and TLB resources.

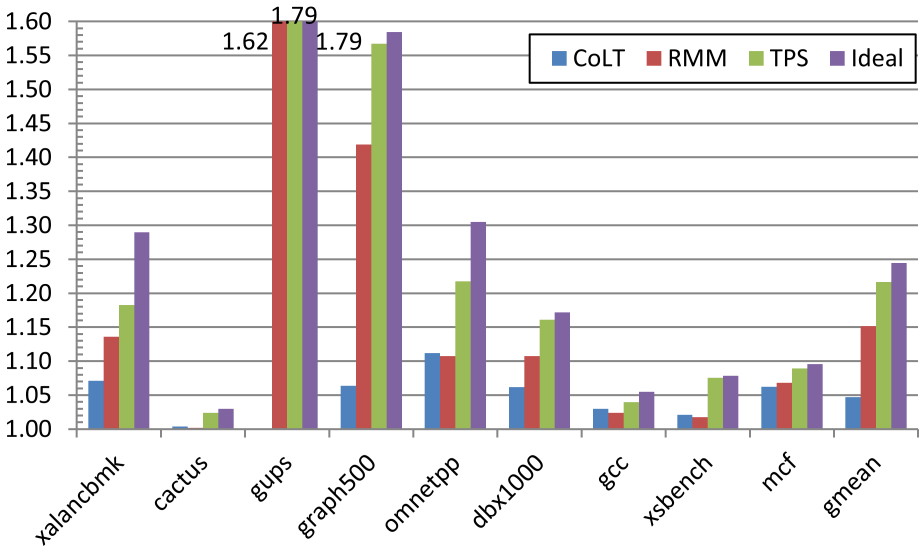

Figure 3.17: Speedup - Native (SMT)

TPS is able to achieve an average performance improvement of 21.6%, as compared to 15.2% for RMM and 4.7% for CoLT, which is 97.7% of the maximal ideal savings for TPS. Some relative values (e.g., for GUPS) are lower for SMT compared to running alone. While the absolute number of cycles spent page walking increased when running under SMT, some applications experienced a greater magnitude of total slowdown from the core resource sharing. This resulted in the page walk overhead representing a smaller percentage of the total execution cycles.

3.4.2.5 Fragmentation

I performed a study to evaluate the impact of external fragmentation on TPS. I examined the state of physical memory on our heavily loaded test server with an uptime of months running Linux kernel version 3.10, with both Transparent Huge Pages and Memory

Compaction enabled and set to `always`. Free memory utilization on the test server was raised to allow just enough for our benchmarks to run (less free memory would cause an out of memory error, which is experimentally uninteresting). Using `/proc/buddyinfo` and `/proc/pid/pagemap`, I identified how much free memory contiguity was available. Figure 3.18 shows what percentage of free memory could be used by single page sizes ranging from 4 KB to 16 MB. Each bar represents what percentage of free memory could be used if only that bar's single page size was used for all allocations. Because 4KB is the smallest possible page size, coverage at this size must be 100%. The key takeaway is that even on a heavily loaded and fragmented system, significant intermediate levels of contiguity exist that can be leveraged by TPS, while only a very small portion of memory contiguity is exclusive to the existing coarse-grained page sizes.

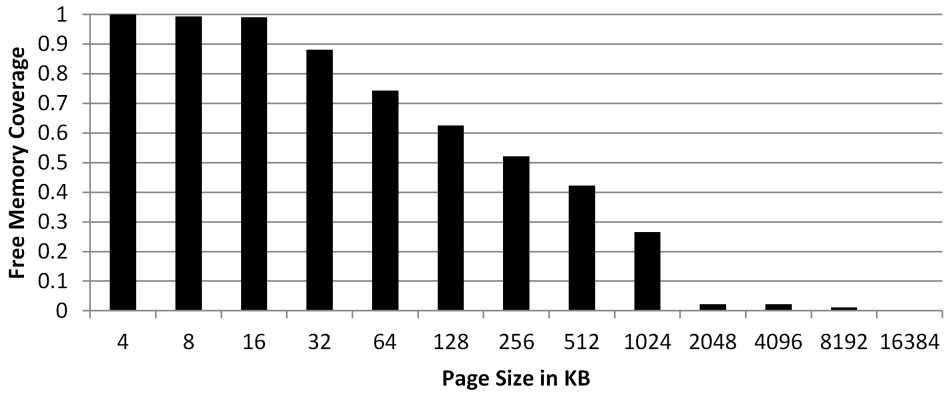

Figure 3.18: Free Memory Coverage by Various Page Sizes

Using the dumped free memory state as input to my PIN virtual memory simulator, I evaluated the potential of TPS under high fragmentation. Figure 3.19 shows that even under these conditions, TPS can attain significant reduction in TLB misses. No memory compaction takes place throughout the simulation. Note that GUPS sees minimal benefit from TPS under high fragmentation conditions. Due to the random access memory behavior of this benchmark, even intermediate page sizes provide limited benefit when the

memory accesses have almost no spatial locality. For other similarly large memory footprint benchmarks like XSBench and Graph500, significant reduction in TLB misses occurs because these benchmarks exhibit some locality in memory references. For long running, large memory benchmarks like GUPS, performing memory compaction at initial allocation time or incremental guided memory compaction over time would help TPS incrementally grow page sizes and reduce TLB misses. As shown in Figure 3.18, even when memory compaction is running, significant memory contiguity at non-coarse grained page sizes exists that TPS would be able to take advantage of.

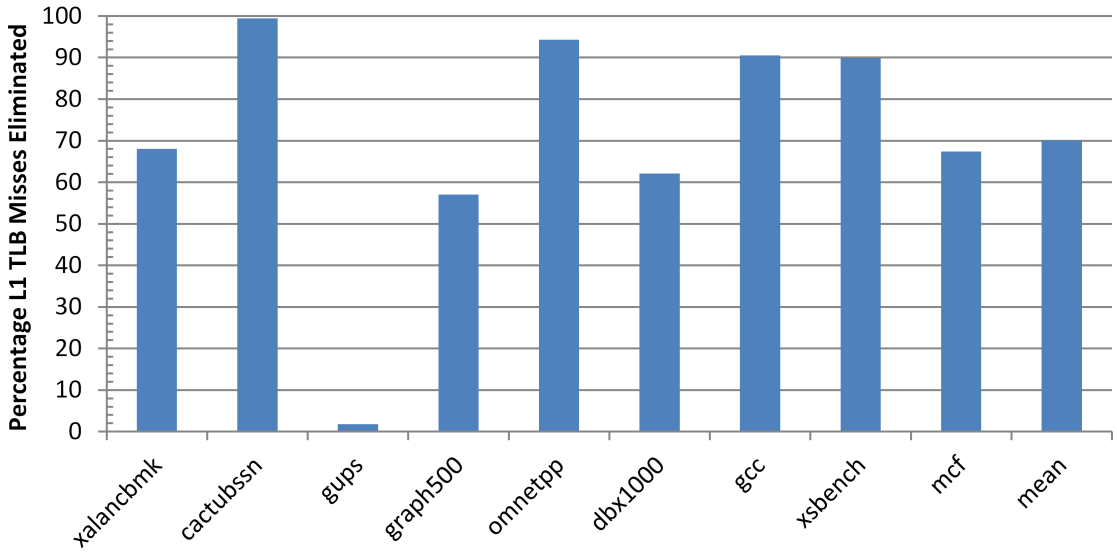

Figure 3.19: L1 DTLB Misses Eliminated

3.4.2.6 System Time

TPS increases the OS allocator complexity which can affect application runtime. Figure 3.20 shows the system time percentage (relative to total execution time) of the workloads. OS allocator work in these memory intensive workloads is very low relative to total execution. The average system time percentage is 0.16%. Even an unrealistic 10x increase in system work due to TPS changes would not cause significant application slowdown.

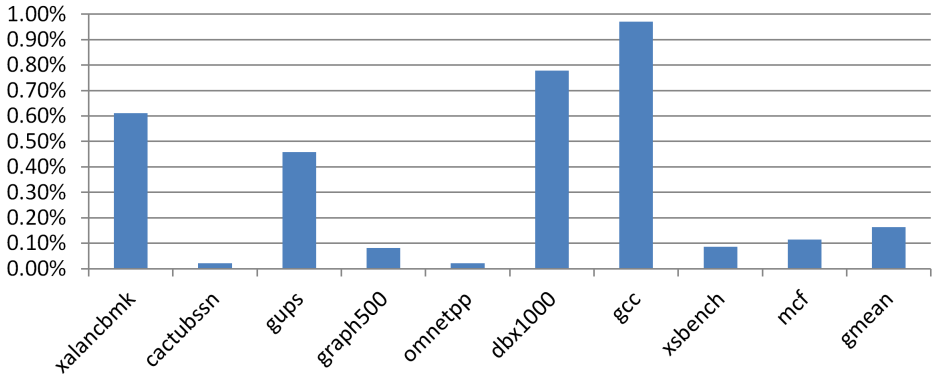

Figure 3.20: Percentage of Total Execution Time Spent in System

3.4.2.7 TPS Created Page Sizes

To minimize the tradeoff between bloat and TLB pressure, TPS allows for pages sized at every power-of-two. I investigated the actual runtime utilization of each page size across the benchmarks. Figure 3.21 shows the results. Each point on the line represents how many pages were in use by that application at the particular page size shown on the x-axis. Each workload utilizes nearly all available page sizes. All benchmarks tend to have higher counts of the relatively smaller page sizes because of the conservative page promotion policy. Even with these higher counts of smaller page sizes, TPS is still able to achieve significant reduction in L1 TLB misses, as shown in the previous sections.

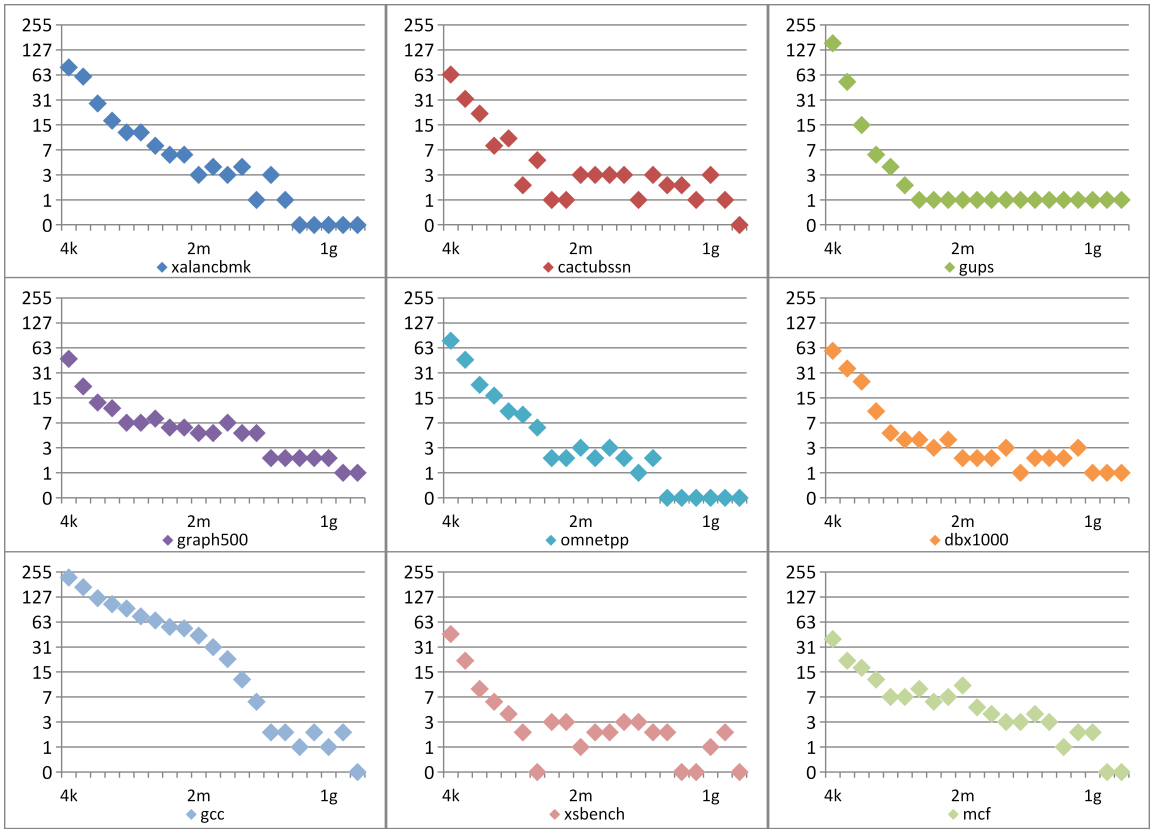

Figure 3.21: Per Benchmark Page Size Counts

3.4.2.8 TLB Storage Overhead

TPS introduced TLB storage overheads for tracking the page sizes within each individual TLB entry. For evaluating storage costs, I compare to the Intel Skylake TLB baseline, details in Table 3.7.

TLB	Entries	Ways
L1: 4k	64	4
L1: 2m	32	4
L1: 1g	4	4 (fully)
L2: 4k/2m	1536	12
L2: 1g	16	4

Table 3.7: Intel Skylake TLB Parameters

TPS saves some TLB costs by consolidating the L1 2M and L1 1G TLBs into a single fully-associative TPS TLB. I also compare to the storage overhead of CoLT and RMM. Table 3.8 shows the overhead (shown as percentage increase in terms of bits of storage), broken down by level in the TLB hierarchy.

TLB	CoLT	RMM	TPS
L1	7.37%	0.00%	6.08%
L2	7.49%	5.75%	3.70%
Total	7.48%	5.41%	3.85%

Table 3.8: TLB Storage Overhead: Percentage Increase Over Intel Skylake Baseline

Overall, TPS has less storage overhead than either CoLT or RMM. TPS uses less storage than CoLT at both levels of the TLB hierarchy. Because RMM introduces a Range TLB which is accessed in parallel with the L2 TLB, the mechanism requires no changes to the L1 TLB.

Introducing the access/dirty bit vectors to TPS results in additional storage overhead. Because the vector access is decoupled from the translation access (as detailed in Section 3.3.1), this storage should have no effect on translation latency. The access/dirty bit vectors exhibit sticky behavior and the OS must perform a TLB shootdown when they are cleared (as with the existing A/D bits). Thus, the L2 TLB does not need to track individual vectors. The cached vectors can be cleared at L1 TLB entry load time. To further reduce vector updates, the L2 TLB can maintain two bits (one for each of A/D) specifying the entire vector has previously been set. Assuming eight bits for each vector, the TPS total TLB storage overhead increases to 6.48%, still less than CoLT.

3.5 Summary

I have shown that current coarse grained page sizes and TLB limitations are insufficient to deliver scalable, high-performance virtual memory translation. I designed Tailored Page Sizes to allow support for pages of any power-of-two size larger or equal to the base page size. TPS requires small changes to hardware and small improvements to operating system software to, at no additional memory cost, significantly reduce L1 TLB misses and page walk memory references, and improve TLB reach.

Chapter 4

Larger Base Page Size

Increasing the base page size provides new opportunities for performance and scalability improvements, but also presents many challenges. Consider that x86 has used the 4KB page size since 1985 [34], with physical memory capacities of 8MB not uncommon. Today, even client devices on the lower end of total memory capacity, e.g., cell phones, have upwards of 4GB of physical memory. Upcoming non-volatile DIMMs will only further increase main memory capacity (e.g., Intel's Optane DC Persistent Memory Module [40]). Despite the 512x increase in total memory capacity, the base page of 4KB remains unchanged. I propose increasing the base page size to 256KB to improve performance and scalability for future processors.

4.1 Benefits

4.1.1 L1 Caches

The larger base page size provides many benefits. It is well known [71, 84] that current L1 cache designs are limited in total capacity to the base page size times the associativity. A high-performance L1 cache design following this pattern can be quickly indexed because all of the cache index bits fit within the page offset field of the address, bits which are not translated. Thus, the virtual to physical translation via the TLB can proceed in parallel with the cache indexing operation. By the time tag bits are available from the cache to compare, the translation should be complete. Figure 4.1 illustrates the process.

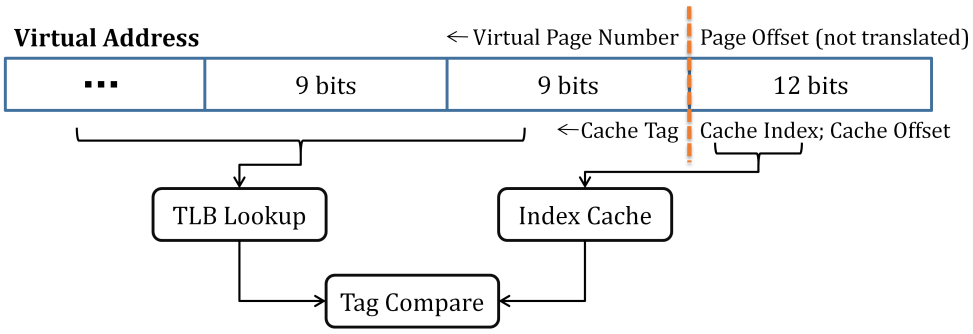

Figure 4.1: Parallel Cache Index/TLB Lookup

Due to this limitation, current processors have been using L1 caches with 32KB total capacity and 8-way set associativity, meaning 4KB per way. The 256KB base page size would enable up to 256KB per way without breaking the parallel cache indexing and TLB translation operations. CACTI [57] results in Figure 4.2 below show that larger cache sizes (e.g., 64KB, 4-way, and nearly 256KB, 4-way) achieve the same latency as current 32KB, 8-way designs.

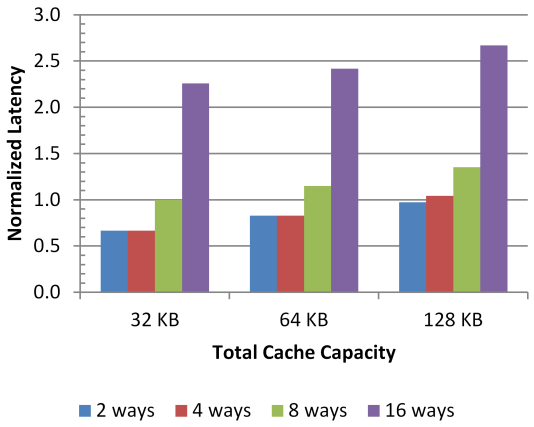

Figure 4.2: Cache Latency From CACTI; Normalized to 32KB, 8-Way Baseline

Similar results have been reported by Zheng et al. [84]; they show that performance improvement can be obtained from reducing associativity by two approaches: maintaining the same size cache, but reducing latency (compared to the 32KB, 8-way baseline); or,

increasing total cache capacity while maintaining identical latency (compared to the 32KB, 8-way baseline).

4.1.2 Memory Dependence Checking

When performing memory dependence checking, optimizations first looking at the page offset bits are possible [27]. If a younger load's page offset field has no overlap with an older store's page offset field, it is impossible for the younger load to depend on the older store. If the page offset fields of these memory operations do overlap, the load may depend on the store. This property holds because the page offset field is not translated and pages are aligned to page boundaries. Figure 4.3 shows an example.

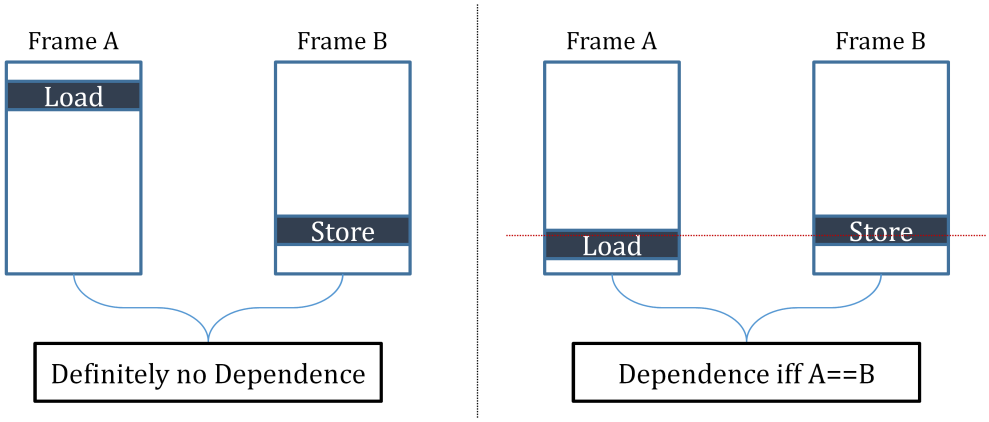

Figure 4.3: Page Offset Memory Dependence Checking

A possible design approach is to force the load to wait until the store's data is available so the physical addresses are known and store-to-load forwarding can be performed. If the load waits for the store, but they did not actually refer to overlapping memory locations, this is a false dependency. With the 12 bit page offset field at the 4KB page size increased to an 18 bit page offset field at the 256KB page size, this mechanism can be improved with false dependencies reduced. Figure 4.4 shows what percentages of all load instructions

exhibit false dependencies when the 4KB and the 256KB page sizes are used. Figure 4.5 shows additional data for the Spec06 benchmarks.

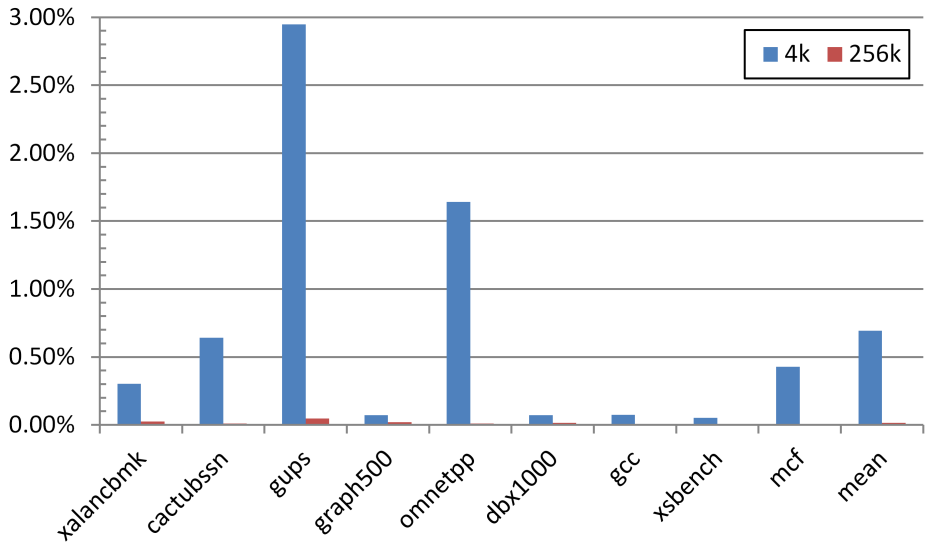

Figure 4.4: Percentage of Dynamic Loads That Experience a False Dependency Given Page Size

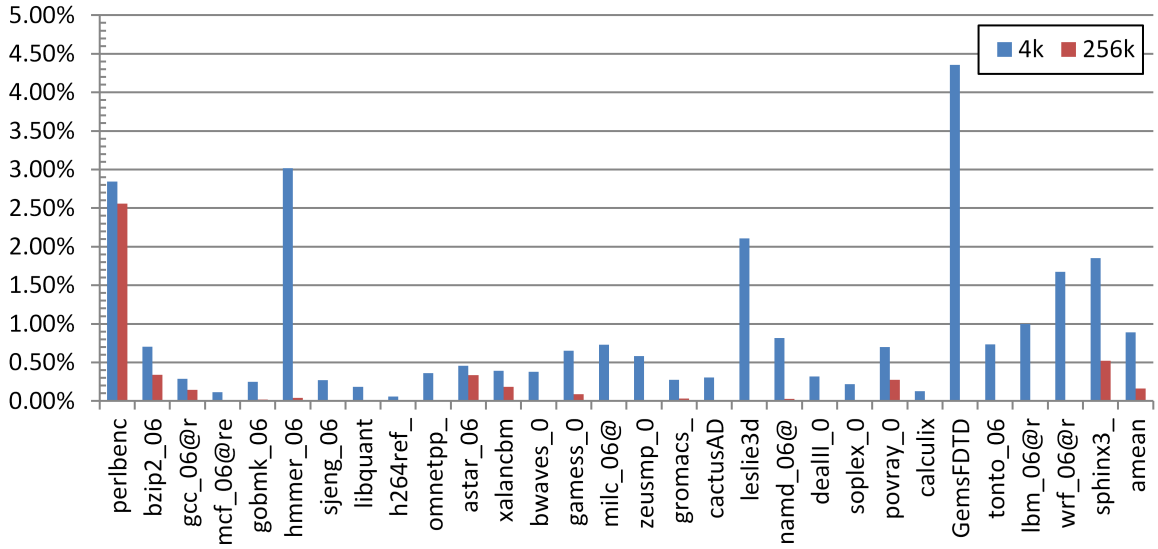

Figure 4.5: Percentage of Dynamic Loads That Experience a False Dependency Given Page Size: Spec06 Benchmarks

4.1.3 Page Table Walks

Like with tailored page sizes, the 256KB larger base page size also helps increase TLB reach. In applications using many base pages, a limited capacity base page size TLB will increase in reach by 64x, the base page size difference.

The hierarchical radix tree page table approach (e.g., x86-64, ARM) typically limits page table sizes to the base page size. With this approach, the number of PTEs in a page table is given by page size divided by PTE size. In x86-64 and ARM, this results in 512-entry page tables at the 4KB base page size. Further, the 48-bit virtual address is divided into four 9-bit fields for the four levels of page table, and a 12-bit page offset field ($4 \cdot 9 + 12 = 48$). At the increased 256KB base page size, this would allow for 32K-entry page tables. This allows the 48-bit virtual address to be divided into two 15-bit fields for the now two levels of page table, and an 18-bit page offset field ($2 \cdot 15 + 18 = 48$). The 4KB base page size results in up to four page walk memory accesses when running non-virtualized, and up to 24 page walk memory accesses on a virtualized machine. The 256KB base page size allows us to reduce this to up to two page walk memory accesses when running non-virtualized, and up to 8 page walk memory accesses on a virtualized machine. Recall the four level page walk process with a 4KB base page, shown in Figure 4.6. Figure 4.7 shows the non-virtualized page walk process with a 256KB base page size.

When the 48-bit virtual address space expands up to 64 bits, every 9 additional bits of virtual address adds an additional level of page tables; this is an ultimately unscalable solution in terms of page walk memory accesses. The 256KB base page size enables only three levels of page table for up to 63 bits of virtual address space.

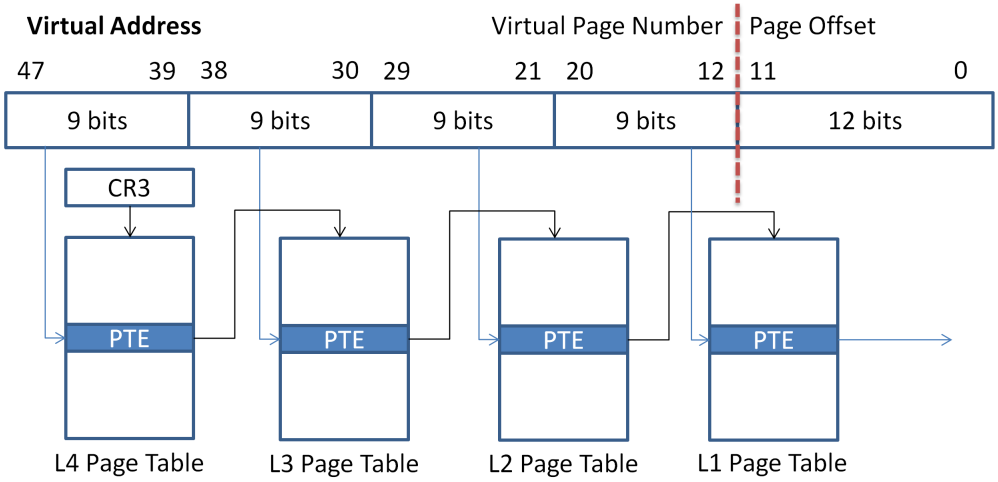

Figure 4.6: Page Walk With 4KB Base Page

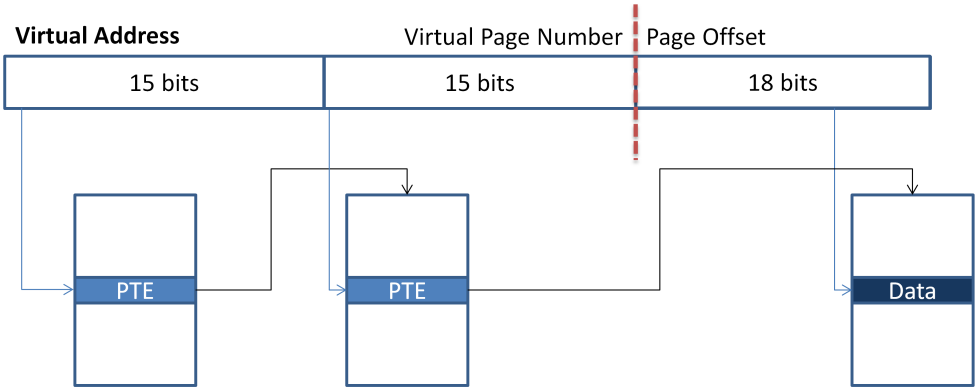

Figure 4.7: Page Walk With 256KB Base Page

Page table memory overhead is reduced with the increased base page size. Using 4KB base pages, mapping 1TB of physical memory could require more than 2GB of page tables. Mapping 1TB with 256KB base pages only requires closer to 32MB of page tables.

4.1.4 Prefetching

Other microarchitectural improvements may be possible with a larger base page size. For example, prefetching across page boundaries is difficult due to the cache hierarchy using physical addresses. Because a data structure that is contiguous across virtual pages may not be contiguous across physical frames, prefetching using the physical address may not refer to the subsequent element in the data structure when crossing the page boundary. In addition, a cross-page stream may not be able to adequately train the prefetcher when pages are non-contiguous. With the 256KB base page size, page boundaries will be encountered by the prefetcher less frequently. I leave the exploration of the prefetcher to future work.

4.2 Challenges

Increasing the base page size presents many challenges, independent of the particular larger size chosen. I discuss all of the challenges and my proposed solutions in the following subsections.

4.2.1 Backwards Compatibility

Backwards compatibility is always a significant issue when making ISA changes. In many cases, software can be hardcoded for the 4KB base page size, or untested on other page sizes. The simple approach to offer a smoother transition period is to provide a mode change bit that chooses between the 4KB base page size and the 256KB larger base page size, as in ARMv8 page size granules [4]. However, this limits the adoption of the potential benefits gained by moving to the larger 256KB base page size. If a microarchitecture cache design assumes the 256KB base page size will be in effect, then moving to a new

processor while still using the 4KB base page size could cause unacceptable performance regression. Assuming L1 cache designs were improved from 32KB capacity with 8 way set associativity, to 64 KB capacity with 4 way set associativity, operating in the 4KB mode presents a cache access problem. The straightforward solutions are to increase the L1 access latency by first waiting for TLB translation for every access, or to only utilize a fraction of the cache sets, limited by the 12 page offset bits. Neither is appealing.

Prior work [28, 54, 51, 84] has already proposed and evaluated physical address predictors that predict a memory access's physical address concurrently to address generation and prior to the TLB access. I propose leveraging the SIPT predictors from Zheng et al. [84] when in 4KB mode to enable no loss in cache access time when the prediction is correct. When operating in 256KB mode, the predictors are unnecessary and will be disabled to save energy. With a 64KB, 4-way cache, only 2 bits of cache index need to be predicted (the total size of 1 way is 16KB, which is 4 times the 4KB page size). The tag comparison still uses the result produced by the TLB translation.

4.2.2 Latency

Increasing the base page size will adversely affect latency in many scenarios. Page swapping, application startup, and allocation latencies may all increase. The current trend in ever-growing physical memory capacity reduces the frequency and importance of page swaps, and some data centers already prefer to run with swapping disabled [8, 48].

In usage scenarios where swapping is still critical, I propose leveraging recently available non-volatile memory express (NVMe; storage over PCIe bus) technology [38] to reduce the swap latency. NVMe technology provides higher density at higher latency than DRAM, but lower latency at lower density than flash-based solid-state drives (SSDs).

Intel currently offers 3DXPoint NVMe products [38] that can be used as a cache for the operating system and the most commonly used programs and files to improve latency. My proposal is to partition this NVMe drive into two; the NVMe will still primarily be used as an OS and file cache, but a small portion will be reserved for the swap, eliminating the need to swap to the SSD or hard disk drive (HDD). For example, a 64GB NVMe drive could be partitioned into 60GB for cache, and 4GB for swap. Figure 4.8 shows the average and tail latency of a Samsung 850 Evo Flash SSD as compared to an Intel Optane 800P NVMe drive for IO sizes ranging from 4KB to 1MB. The 128KB size on NVMe is very close to flash at 4KB, while 256KB NVMe average latency remains within an order of magnitude.

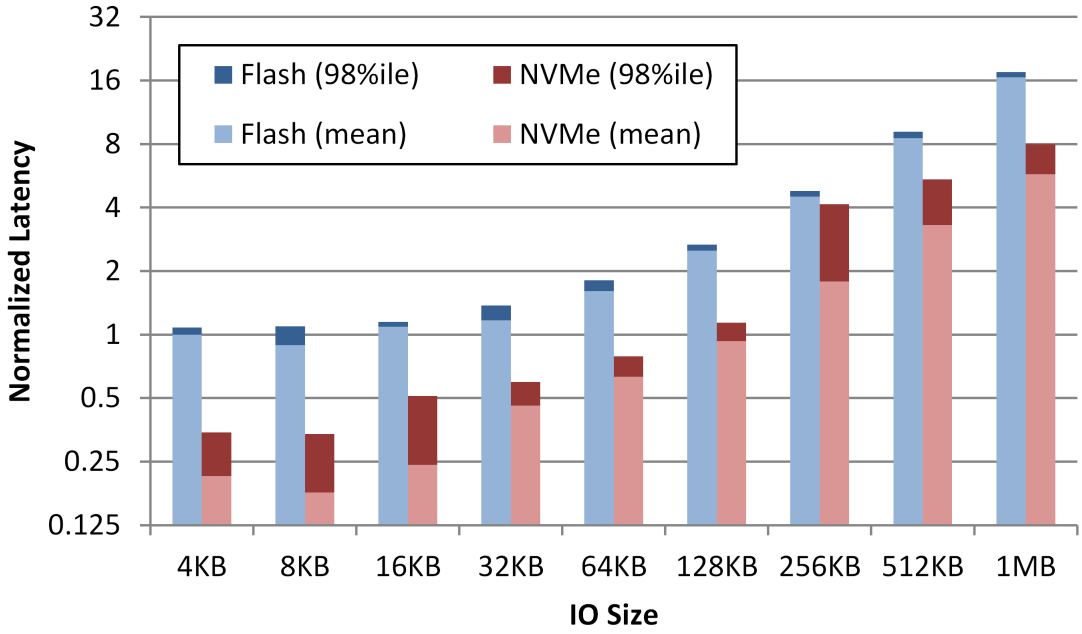

Figure 4.8: Flash vs. NVMe IO Latency (Normalized to Average Flash 4KB Latency)

To address application startup and allocation latencies, I propose variable length base page sectoring. Sectored TLBs have been previously proposed [75, 46]; however, the key difference is my support for optionally sectoring the smallest page size using one of

many sector sizes. The 256KB base page size results in six fewer page number bits; these six fewer page number bits provide six extra bookkeeping bits that can be used in the PTE. Three of these six extra bits are enough to specify sector size. Bit pattern 000 specifies no sectoring used, while the remaining patterns allow for power-of-two sector sizes: from as many as 64 4KB sectors to as few as two 128KB sectors.

Sectors are always aligned within the parent base page. For example, the first 4KB of a 256KB virtual page correspond to the first 4KB of the 256KB physical frame.

The individual sectors within a page require bookkeeping bits. The current page tables and PTEs do not have room for additional sector bookkeeping bits. These bookkeeping bits will be kept in a new *sector page table*. With sectoring disabled, the base page's PTE points to the page containing the data. With sectoring enabled, the base page's PTE points to the sector page table. The sector page table contains the (up to) 64 valid bits, one for each individual sector, and the page's page frame number. Figure 4.9 illustrates the updated PTE and page tables.

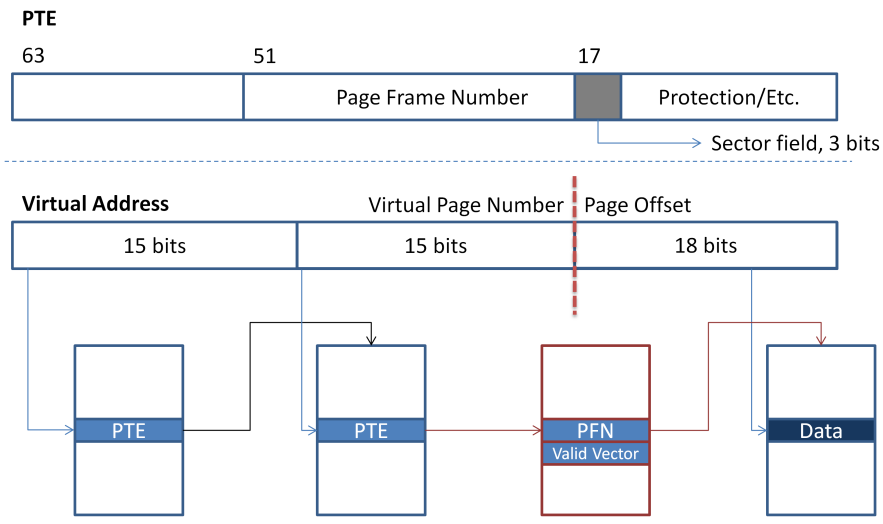

Figure 4.9: Page Walk and PTE for Sectored 256k Page

The TLBs will cache the sector valid bits. To avoid an increase in TLB lookup latency, these bits are decoupled from the rest of the TLB entry. If the primary TLB entry identifies the entire base page as a valid PTE, then the core can speculatively assume the accessed sector is also valid and the memory access will be allowed. Concurrently with the rest of the cache access pipeline, the accessed sector will be verified as valid. This decoupling prevents the sectoring operation of the TLB from having impact on translation latency. In the common case, the sector should also verify as valid; in the uncommon case, a page fault occurs which means the minor additional latency from misspeculation is largely irrelevant. In light of the recent Meltdown [50] and Spectre [47] attacks, it is critical that although the sector verification is decoupled, data from invalid sectors must not affect what cache lines are present in the cache hierarchy nor forward to subsequent instructions consuming the data.

To determine a good minimal sector size, I studied how allocation latency breaks down into setup time (syscall, page table setup, page fault) versus actual initialization of the page (e.g., clearing memory for a new anonymous mapping). The sum of setup time and memory initialization is the total allocation latency. Figure 4.10 shows the (computed) allocation latency results for sizes from 4KB to 1MB.

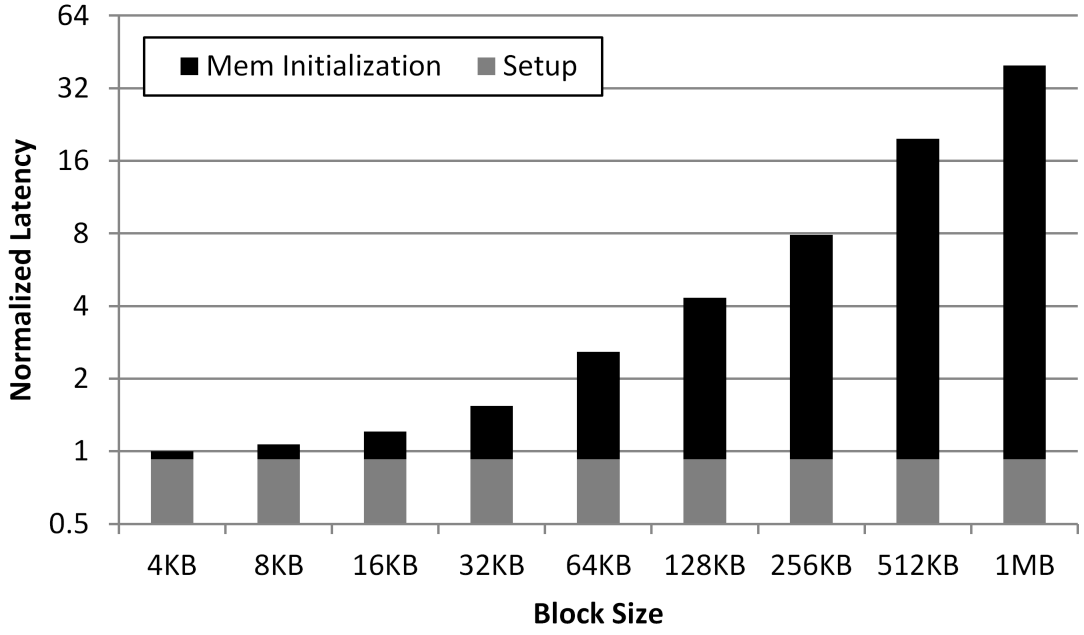

Figure 4.10: Allocation Latency Breakdown (Normalized to Total Allocation Latency at 4KB Block Size)

I timed and averaged across millions of 4KB mmap calls created as copy-on-write of a zero page to identify setup time. I then timed clearing already mapped memory for sizes from 4KB to 256KB to identify memory initialization time. Results show setup time significantly outweighs memory initialization time for the current 4KB size.

To achieve latency parity with 4KB base pages, application startup and new allocations could always initially begin with 4KB sectoring enabled; however, the previous results show that a 32KB minimal sector size may be significantly more appealing due to

a better latency balance between setup and memory initialization. When an entire 256KB page is filled out, sectoring can be disabled for the page. As technology improves, OS software may begin to prefer larger sectors until sectoring for allocation latency is ultimately phased out. Further, utilizing recently proposed techniques like RowClone [68] can significantly reduce allocation latency.

4.2.3 Fragmentation

The larger base page size potentially increases internal fragmentation. In many cases, I posit that continued growth in memory capacity enables applications to sacrifice some memory to internal fragmentation in order to obtain the other benefits of the increased base page size. For example, closer to the introduction of 4KB pages, total memory capacity approached 8MB. The maximum per allocation loss to internal fragmentation is 0.049% (i.e., 4KB/8MB) of the total memory capacity. With the base page size of 256KB and main memory capacity of at least 2GB, the maximum per allocation loss to internal fragmentation is only 0.012% (i.e., 256KB/2GB) of the total memory capacity.

The page cache and applications like browsers and source code trees may contain many small files. For these kinds of applications, it is reasonable to argue that the loss to internal fragmentation with the increased base page size may be too great. My solution is to allow memory management for files at a finer granularity than the increased base page size. Virtual address space is still significantly more plentiful than physical memory. My solution is to use the variable length base page sectoring proposed in Section 4.2.2. I can identify a 256KB region of virtual memory that is free, only activate the sectors within the base page corresponding to the file's location in the frame, and map the virtual page to the frame containing the file, thus preventing additional loss to internal fragmentation for these

cases. Figure 4.11 illustrates an example. An additional sector bit vector is needed for the OS to specify which sectors are mappable versus unmappable.

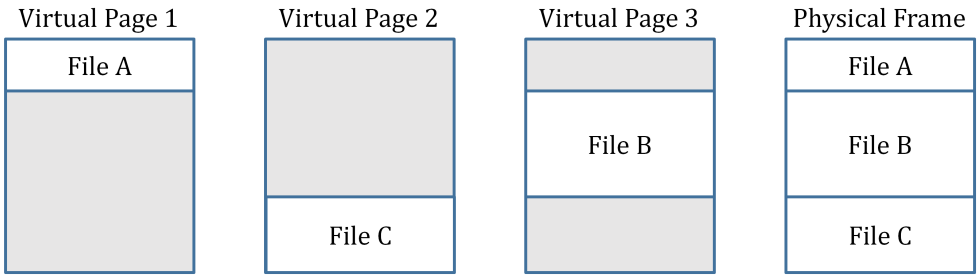

Figure 4.11: Multiple Small Files in a Frame With Sectoring

4.2.4 Fine-Grained Metadata Tracking

Increasing the base page size results in increasing the granularity of reference, modification, and protection tracking. In some scenarios, finer grained tracking may be desired. Virtual machine live migration repeatedly tracks modified pages to ship from the migration source to the migration destination. This process repeats until memory is largely coherent at the destination, at which point execution moves to the destination. Tracking modified data at a larger granularity may have significant impact on migration time. Memory mapped file I/O could be similarly affected, resulting in significantly more I/O writes to storage. This could be particularly problematic for SSDs with more limited lifetime write wear.

To handle these problems, I propose augmenting my variable length base page sectoring with optional reference, modification, and protection bits. Additional bookkeeping bits in the PTE specify whether sector level metadata tracking is enabled. The sector page table contains metadata bit vectors that are cached by the TLB. Like the sector valid bits, these can be decoupled from the rest of the TLB to minimize impact on TLB latency.

4.2.5 Granularity of Successive Available Coarse-Grained Page Sizes

Increasing the base page size to facilitate fewer levels of page table results in larger intervals between available coarse-grained page sizes. For example, the next larger size with 4KB base pages is 2MB (a 512x increase in page size). The next larger size with 256KB base pages is 8GB (a 32768x increase in page size). Tailored Page Sizes will allow any power-of-two sized page in between the coarse grained page sizes to be created, solving this problem. I call applying TPS to the larger base page size TPS+.

4.2.6 Memory Deduplication

Memory deduplication (e.g., Kernel Samepage Merging [17]) and copy-on-write (COW) pages could be adversely affected by the larger base page size. Larger pages could mean COW latency is increased and memory savings are reduced. Larger pages reduce the likelihood of finding identical pages to deduplicate. Page Overlays [69] are a previously proposed solution to facilitate fine-grained COWs with a technique called overlay-on-write. This mechanism can be used in conjunction with my larger base page size to provide the OS with the option to improve memory deduplication and COW performance when desired. I leave further exploration in this area to future work.

4.3 Evaluation

4.3.1 Methodology

See Chapter 3, Section 3.4.1 for an overview on methodology. I list further methodology differences or additions in this subsection.

To evaluate the microarchitectural improvements possible via the larger base page, I use an execution-driven, cycle-based in-house simulator of an x86 superscalar out-of-

order processor. Functional simulation is based on PIN. The simulator faithfully models core microarchitectural details, the cache hierarchy, wrong-path execution, and includes a detailed DDR3 memory system model. Evaluation is performed on a single-core system.

To evaluate the virtual memory translation benefits of the larger base page, I augmented my PIN-based virtual memory simulator with support for the larger base page and two dimensional translation from guest virtual address to host virtual address to machine physical address.

4.3.2 Results

4.3.2.1 Microarchitectural Evaluation

I explored the benefits of increased cache size, made possible by the larger base page. Figure 4.12 shows the speedup of a 64 KB, 4-way set associative and a 128KB, 4-way set associative cache compared to the 32 KB, 8-way set associative baseline. Cache access latency was held constant across all experiments. For very large, low locality working sets like in GUPS, a small increase in L1 cache capacity has negligible impact on performance. XSBench actually experiences a very minor performance degradation due to the reduced associativity. Average performance improvement for the 64KB configuration is 1.6%.

Figure 4.13 shows the speedup of 256kB granularity (and perfect) dependence checking compared to the 4kB granularity baseline. In general, as previously shown, dynamic loads experiencing false dependencies are a relatively small percentage of total program dynamic loads, so performance change is generally relatively small. Average improvement is 0.6%. Mcf suffers a minor performance hit from 256kB to perfect due to wrongpath effects; the delay of wrong path falsely dependent loads prevented a small amount of useless memory requests/cache pollution.

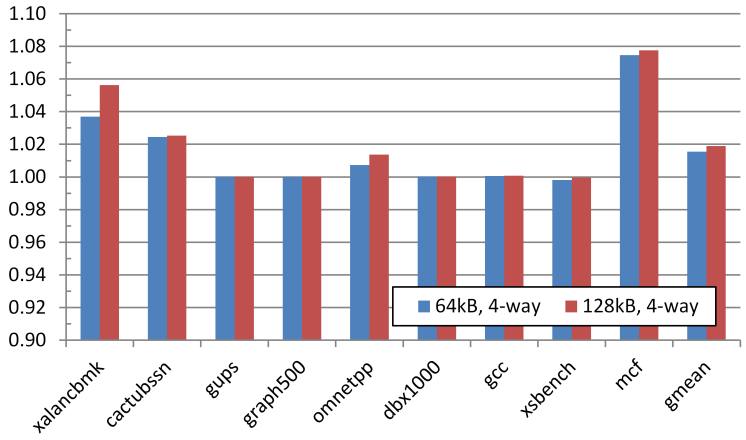

Figure 4.12: Speedup of Larger Cache Configurations Compared to 32KB, 8-Way Baseline

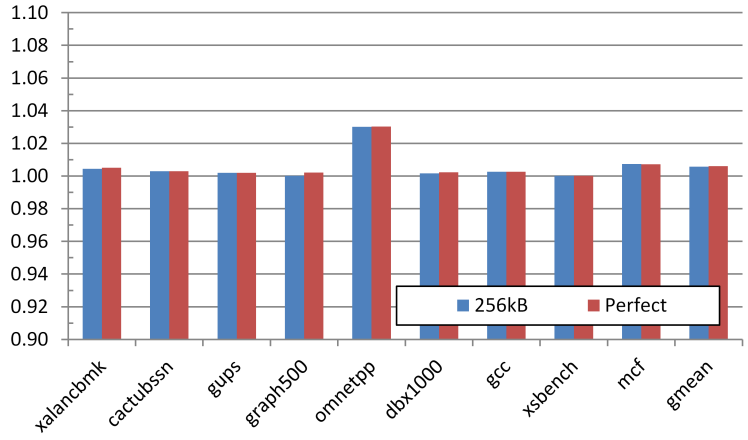

Figure 4.13: Speedup of Improved Dependence Checking Compared to 4kB Granularity Baseline

Figure 4.14 show the speedup of both techniques. The evaluated configuration uses a 64KB, 4-way cache with 256KB granularity dependence checking, compared to the 32KB, 8-way cache with 4KB granularity dependence checking baseline. In general, performance improves from both sources. Average improvement is 2.0%.

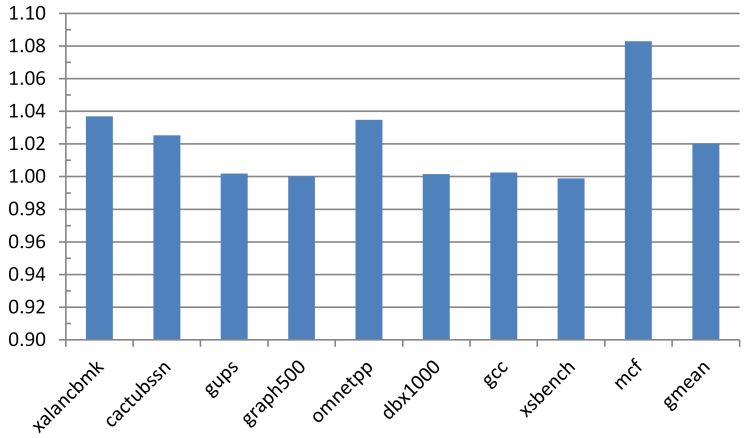

Figure 4.14: Speedup of 64KB, 4-Way Cache with 256KB Granularity Dependence Checking Compared to 32KB, 8-Way Cache with 4KB Granularity Dependence Checking

4.3.2.2 TLB Miss Reduction Due to 256KB Base Page

Increasing the base page size to 256KB reduces the TLB miss rate by improving TLB reach. To quantify this reduction, I studied the effect of the larger base page size in my PIN-based virtual memory simulator. Figure 4.15 shows the percentage of L1 DTLB misses eliminated by exclusive 256KB pages as compared to the exclusive 4KB pages baseline.

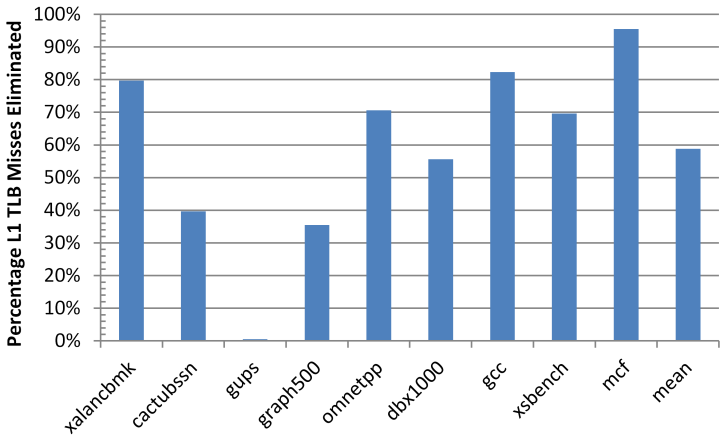

Figure 4.15: 256KB Base Page: L1 DTLB Misses Eliminated (Exclusive 4KB Pages Baseline)

Average reduction in L1 DTLB misses is 58.8%. Two key factors that affect how well the 256KB base page can reduce TLB misses are data locality (spatial/temporal) and working set size. The 256KB base page has almost no affect on GUPS because of its large working set size and random access pattern. Other larger working set size benchmarks like DBx1000, CactuBSSN, and Graph500 exhibit lower reductions in L1 DTLB misses.

Figure 4.16 shows the reduction in page walk memory references for native execution. Average reduction in page walk memory references is 89.5%. Several factors affect the magnitude of reduction. Improved TLB hit rate reduces the number of page walks. In addition, the 4KB base page size uses a four level page table while the 256KB base page size uses a two level page table. This results in the worst case number of memory references being cut in half for each page walk. The MMU caches are more effective with the larger base page size. For example, a 64 entry L2 Page Table cache with the 256KB base page size covers 512 GB of memory. With the 4KB base page size, a 64 entry L2 Page Table cache covers only 128 MB of memory.

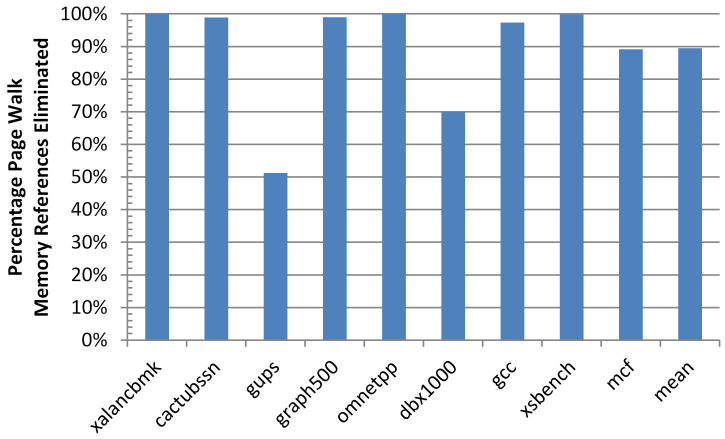

Figure 4.16: 256KB Base Page: Page Walk Memory References Eliminated, Native Execution (Exclusive 4KB Pages Baseline)

Figure 4.17 shows the reduction in page walk memory references for virtualized execution. Due to the two dimensional page table walk, the worst case number of memory references is reduced from 24 for the 4KB base page size to 8 for the 256KB base page size. This greater disparity in number of memory references per page walk results in a larger relative reduction in page walk memory references. The average reduction is 91.5%.

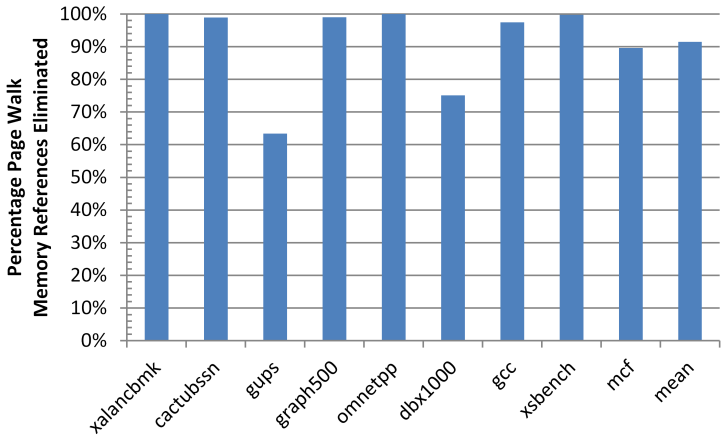

Figure 4.17: 256KB Base Page: Page Walk Memory References Eliminated, Virtualized Execution (Exclusive 4KB Pages Baseline)

4.3.2.3 Tailored Page Sizes Plus 256KB Base Page

I evaluated the 256KB base page size working in conjunction with Tailored Page Sizes (TPS+). Like with TPS, I ran my simulations under the most conservative reservation strategy, requiring 100% utilization of base pages before merging into a larger page (e.g., two buddy 256KB base pages must have both been used before upgrading them into a 512KB tailored page). The baseline for comparison is reservation-based Transparent Huge Pages (THP).

Figure 4.18 shows TPS+'s reduction in L1 DTLB misses compared to THP. The minimum value shown on the y-axis is 95%. Average reduction is 99.9%. Due to the relatively small capacity on the L1 TLB, the THP baseline experiences very high contention and significant numbers of TLB misses which TPS+ nearly eliminates altogether.

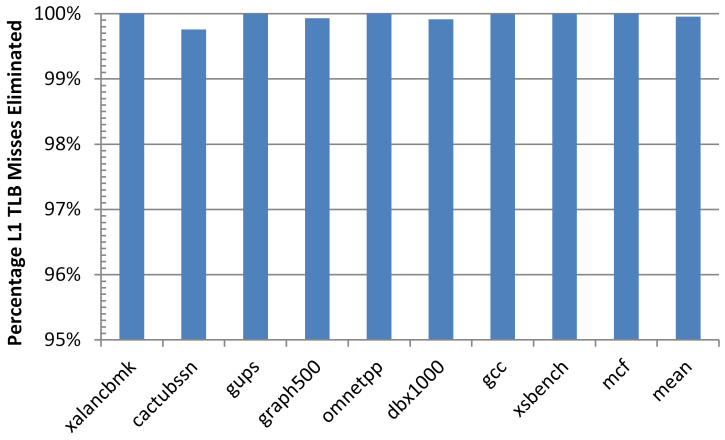

Figure 4.18: TPS+: L1 DTLB Misses Eliminated (THP Baseline)

Figure 4.19 shows the reduction in page walk memory references for native execution. Average reduction in page walk memory references is 99.6%. The large L2 TLB capacity leads to relatively less contention even in the baseline configuration. Tailored page sizes may experience extra page walk memory references due to the alias PTEs. Even so, TPS+ is able to eliminate most page walk memory references.

Table 4.1 shows the hit rate for each level of MMU cache. Hit rates are reported for two configurations: 1) the baseline THP, and 2) TPS+.

In all cases, the actual total number of page walks is significantly reduced with TPS+. TPS+ does not use the L3 or L4 page table caches because the 256KB base page size results in only two levels of page table. The L2PT$ with TPS+ is significantly more effective because each entry covers 8GB (compared to the 2MB for each entry of the L2PT$

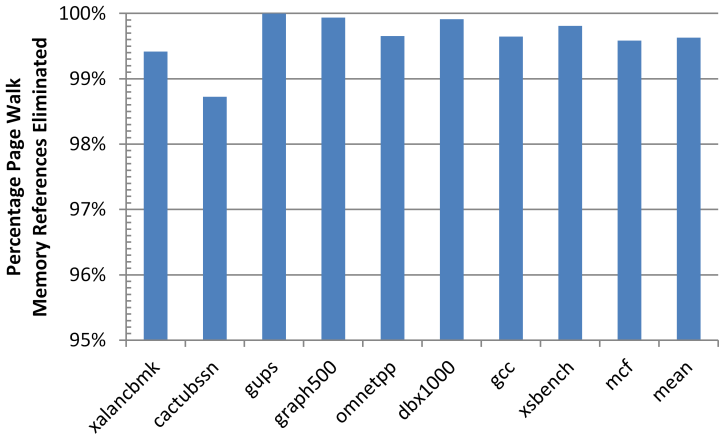

Figure 4.19: TPS+: Page Walk Memory References Eliminated, Native Execution (THP Baseline)

	Baseline (THP)			TPS+
Benchmark	**L2PT$**	**L3PT$**	**L4PT$**	**L2PT$**
xalancbmk	98.6%	99.9%	0.0%	99.8%
cactubssn	72.4%	100.0%	72.7%	100.0%
gups	1.2%	100.0%	69.2%	100.0%
graph500	80.3%	100.0%	33.3%	100.0%
omnetpp	99.9%	98.3%	33.3%	99.7%
dbx1000	3.8%	100.0%	78.6%	100.0%
gcc	58.1%	100.0%	75.0%	100.0%
xsbench	35.8%	100.0%	60.0%	100.0%
mcf	39.4%	100.0%	77.8%	100.0%
Mean	54.4%	99.8%	55.6%	99.9%

Table 4.1: MMU Cache Hit Rates (TPS+)

in the baseline case). For these benchmarks at these input set sizes, the L2PT$ effectively only experiences the compulsory cold misses.

Figure 4.20 shows the reduction in page walk memory references for virtualized execution, assuming host page size is able to match guest page size. Average reduction in page walk memory references is 99.8%. The reduced number of steps in the two dimensional page walk possible via the 256KB base page size enables a greater relative proportion of page walk memory references (compared to native execution) to be eliminated.

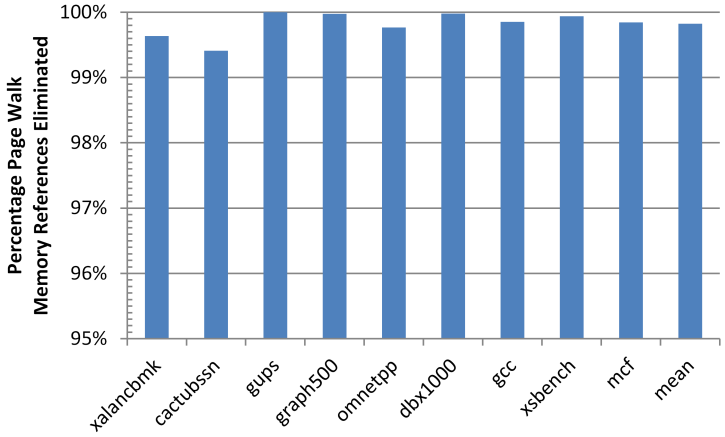

Figure 4.20: TPS+: Page Walk Memory References Eliminated, Virtualized Execution (THP Baseline)

Using the $T = T_{IDEAL} + T_{L1DTLBM} + T_{PW}$ performance model from Section 3.4.2.4 and the previously evaluated TLB miss and page walk memory reference reductions, I approximate the performance benefit of TPS+ resulting from reduced translation latency. Figure 4.21 shows the speedup of TPS+ in three execution scenarios: native execution, native execution with a co-running SMT thread, and virtualized execution. TPS+ achieves average performance improvements of 16.7%, 24.4%, and 59.7%, respectively. These correspond to greater than 99.9% of the ideal benefit (zero TLB misses). The very high speedups of GUPS and Graph500 bring up the average; excluding these two benchmarks results in gmean performance improvement of 25.2% on virtualized execution.

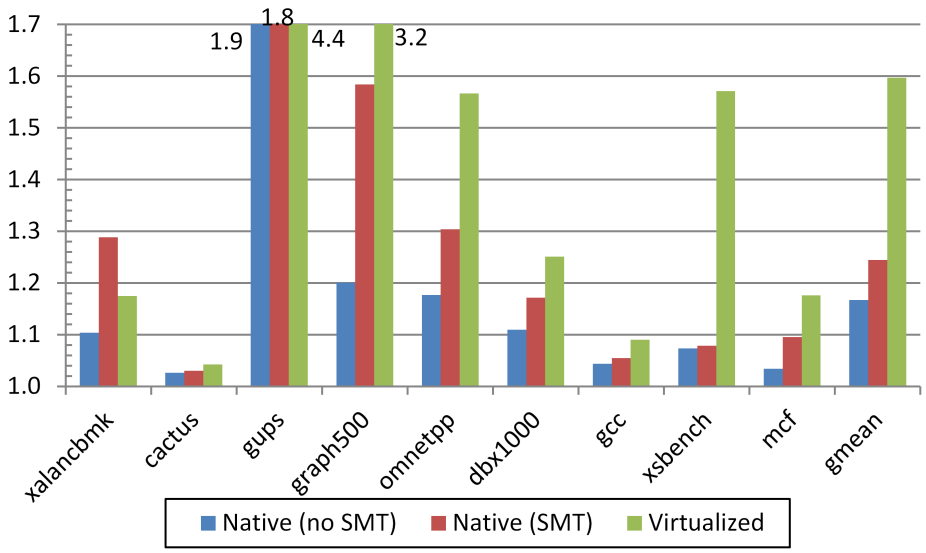

Figure 4.21: TPS+: Speedup (THP Baseline)

As discussed in Section 4.2.3, the larger base page size can result in wasted memory due to internal fragmentation. Figure 4.22 shows TPS+'s increase in memory utilization as compared to the THP baseline. The application's number of sparse and small allocations relative to the application's total working set size is the primary factor in memory loss due to internal fragmentation. Even so, average increase in memory utilization is only 0.72%; the maximum increase is omnetpp with 3.89%, and most applications fall below 0.25%.

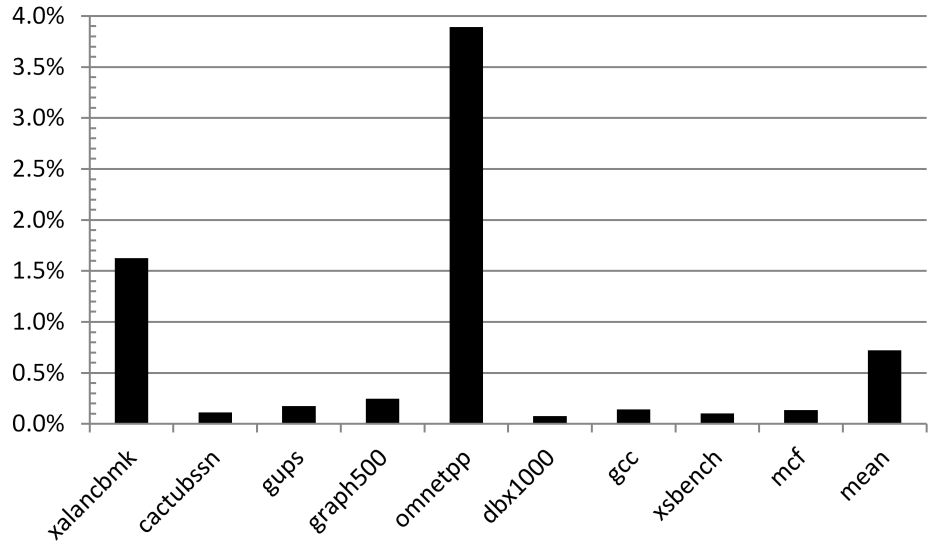

Figure 4.22: Increase in Memory Utilization with TPS+ (THP Baseline)

I investigated TPS+'s actual runtime utilization of each page size across the benchmarks. Figure 4.23 shows the results. Each point on the line represents how many pages were in use by that application at the particular page size shown on the x-axis. Even with the 256KB base page size, each workload still utilizes nearly all available page sizes. All benchmarks tend to have higher counts of the relatively smaller page sizes because of the conservative page promotion policy. The relatively small total number of unique pages is what ultimately enables TPS+ to eliminate nearly all TLB misses, as shown in previous paragraphs.

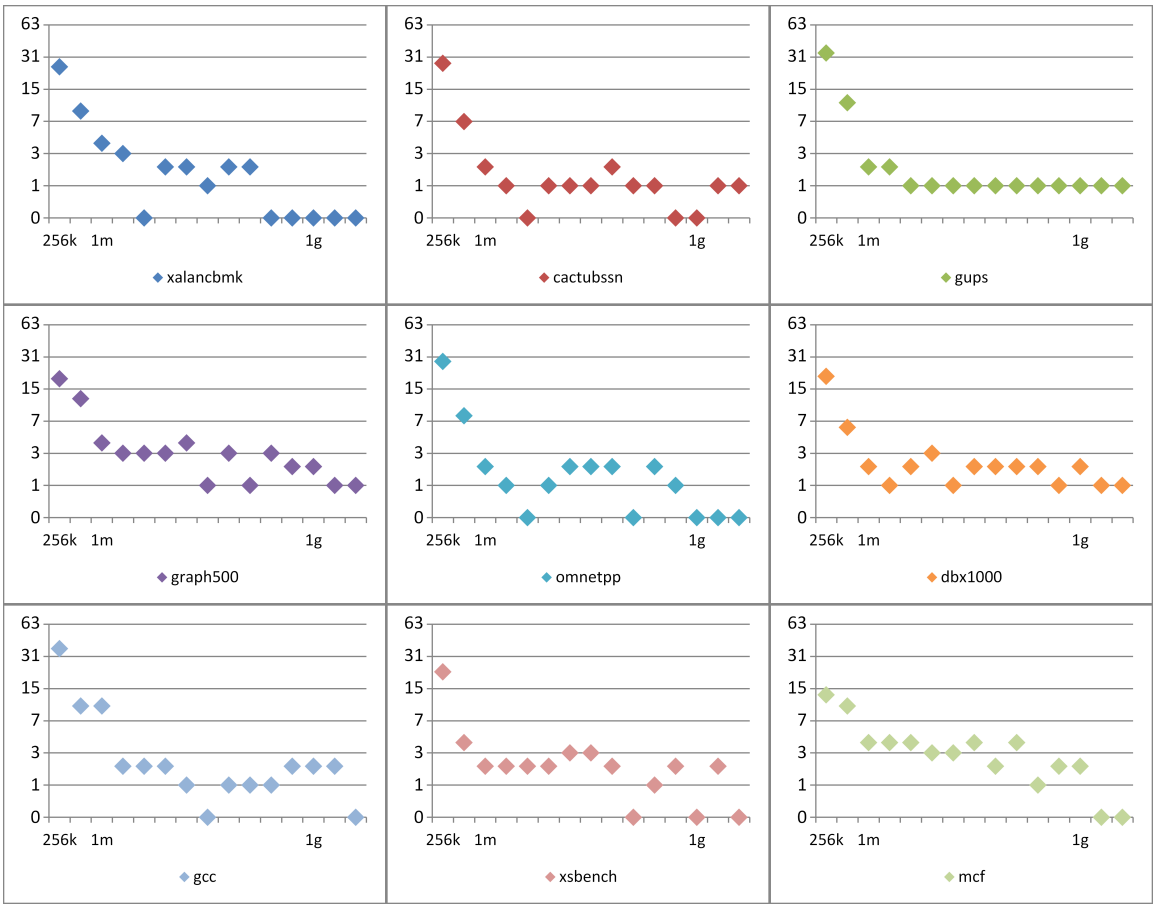

Figure 4.23: TPS+: Per Benchmark Page Size Counts

4.3.2.4 TLB Storage Overhead

To evaluate storage costs, I compare six different TLB configurations to the Intel Skylake TLB baseline as in Section 3.4.2.8. Details of each configuration are in Table 4.2. All configurations use the 256KB base page size, while varying several parameters. To support base page sectoring, each TLB entry may have sector bit vectors (valid, referenced, modified, protection (2 bits)). I report overhead assuming 32KB minimum sector size. Since not all pages will have sectoring activated, storage can be saved by limiting sector bits to a fraction of TLB entries (e.g., 50% or 25% as in C and D). Configurations E and F add TPS support. Configuration F includes TPS access/dirty vectors.

Config	Description
A	256 KB Base Page - No Sector Bits/No TPS
B	256 KB Base Page - Per Entry Sector Bits/No TPS
C	256 KB Base Page - Half Entry Sector Bits/No TPS
D	256 KB Base Page - Quarter Entry Sector Bits/No TPS
E	256 KB Base Page - Quarter Entry Sector Bits/With TPS
F	256 KB Base Page - Quarter Entry Sector Bits/With TPS+TPS A/D Vectors

Table 4.2: TLB Configuration

Table 4.3 reports the results for each configuration, broken down by TLB level. The base 256 KB base page configuration saves on storage due to fewer coarse-grained page size options and fewer VPN bits. Including per TLB entry sectoring bits is moderately expensive, 29.6% as shown in column B. Sectoring bits do not affect translation latency because their access is decoupled from the primary TLB access, as previously described. Only applying sectoring bits to 25% of TLB entries ultimately results in total TLB storage overhead of 8.8% for TPS+.

TLB	**A**	**B**	**C**	**D**	**E**	**F**
L1	-10.8%	13.3%	1.3%	-4.8%	5.5%	16.4%
L2	-6.5%	30.6%	12.1%	2.8%	6.5%	8.4%
Total	-6.7%	29.6%	11.4%	2.4%	6.5%	8.8%

Table 4.3: TPS+ TLB Storage Overhead: Percentage Increase Over Intel Skylake Baseline

4.3.3 Case Study: FireFox Web Browser

To examine the effects of the larger base page size on workloads with many (total and smaller-sized) mappings, I analyzed the virtual memory utilization of the FireFox Web Browser, after running for multiple days with 50 active tabs of various web pages.

FireFox had 1325 virtual memory mappings, with 12.9GB total memory size. For a given base page size, Table 4.4 shows: 1) how many mappings FireFox had smaller than that base page size, 2) the absolute increase in memory utilization if those smaller mappings were grown to the specific larger base page size, and 3) the resulting total percentage increase in memory utilization. For this particular web browser instance, the results show that the 256KB base page size has minimal impact on wasted memory.

Size	Number of Mappings	Increase in MBs	Percentage Increase
8KB	349	1.4	0.01%
16KB	427	4.6	0.03%
32KB	481	11.9	0.09%
64KB	532	27.9	0.21%
128KB	691	69.9	0.53%
256KB	802	166.9	1.26%

Table 4.4: Memory Impact of Larger Base Pages on FireFox

4.4 Summary

I have shown that increasing the base page size to 256KB offers significant room for performance improvement. The larger base page size presents some challenges, which I have addressed with sectoring, fine-grained metadata tracking bits, swapping to NVMe, and other techniques. Together with TPS, the 256KB base page size eliminates more than

99% of TLB misses, resulting in 59.7% average performance improvement on virtualized platforms.

Chapter 5

Related Work

Active research continues in translation latency and virtual memory. Prior work has shown that excessive page walks may significantly degrade performance in applications suffering from limited TLB reach [8, 14, 24, 41, 42, 44]. I have further shown that L1 TLB misses that still hit in the L2 TLB may cause non-trivial performance degradation. I have already discussed the most directly related background information in Chapter 2. I will discuss the related work in this chapter.

5.1 Redundant Memory Mappings

Redundant Memory Mappings (RMM) [44, 45] is the most closely related work to TPS+. RMM leverages arbitrary ranges of contiguous virtual pages that map to contiguous physical frames to provide an alternative range translation mechanism. In the operating system, RMM requires a Range Table to be maintained concurrently with the standard Page Table. Both the Range Tables and the Page Tables map the process virtual address space to physical frames. In hardware, a cache of Range Table Entries (or Range TLB; similar to a TLB being a cache of Page Table Entries) provides an alternative mechanism for the hardware to translate a virtual address to a physical address. Each Range Table Entry (RTE) is similar to a segment descriptor, containing base, limit, offset, and protection information. When a translation misses in the L1 TLB, the L2 TLB and Range TLB are looked up in parallel because both are able to provide the necessary translation. If the

Range TLB provides the translation, the PTE for the page can subsequently be constructed and installed in the L1 TLB.

While both RMM and TPS+ target the same problem, there are many differences between the two approaches. For example, because range translations do not have alignment or size restrictions, it is likely that RMM is more amenable to system external fragmentation. However, I will describe the reasons why TPS+ offers a cleaner solution.

First, RMM proposes always utilizing an eager paging strategy to completely allocate entire contiguous regions. While necessary to facilitate range creation in RMM, this strategy has the important drawbacks of increasing application startup time and allocation latency. In contrast, TPS+ utilizes demand paging with frame reservation as an alternative to eager paging. Since reservation avoids these costs, TPS+ mitigates these issues in situations where they may be particularly problematic.

RMM offers no clean solution to maintain and update page accessed/dirty bits for individual pages within a range and/or entire ranges (and how to deal with consistency between the two if these bits were to be tracked at the entire range granularity). RMM instead proposes setting all of the accessed and dirty bits for all of the pages within a range translation at allocation time. This approach has the effective result of removing any utility of the accessed and dirty bits. In contrast, TPS+ can either maintain the accessed/dirty bits no differently than with current coarse-grained paging techniques, or use the proposed fine-grained tracking bit vector stored within the alias PTEs and cached in the TLBs.

Because RMM has no size and alignment restrictions, looking up the Range TLB requires significantly more logic than a standard TLB lookup. Two comparisons must be performed against the Range Base and Range Limit, followed by the introduction of an addition operation of the Virtual Page Number with the Range Offset. In contrast, TPS+

introduces only a single gate delay to the standard TLB lookup as previously described in Chapter 3. While the necessary Range TLB operations would not be a problem at the L2 TLB level, it is unclear if it would be possible to perform these operations at the L1 TLB level without impacting translation latency for the common case.

Finally, RMM introduces the software complexity of maintaining two trees in parallel to represent the same virtual-to-physical address space mappings (the Range Table Tree and the Page Table Tree). In certain situations, this additional work can present unforeseen complexity with regards to correctness and performance. For example, the current page table structure implements a very subtle fine-grained locking scheme with respect to the four levels of the page table radix tree hierarchy [21]. Originally, a single mutex was used to protect the entire page table tree, but due to performance reasons, this was improved to a two-lock algorithm. Even more recently, even the two-lock algorithm bottlenecked performance in certain cases, so the locking algorithm was further improved to a fine grained algorithm requiring a lock per each level of the page table tree [21]. It is not immediately clear how to apply this locking algorithm to the two tables in parallel in a way that maintains current high performance and correctness. TPS+ requires no changes to the page table locking algorithm, and in general presents a cleaner software engineering solution than RMM.

5.2 Larger Pages

Huge pages or superpages [18, 19, 58, 76] are already utilized in current processors. The prevalent (e.g. Intel and ARM) approaches offer limited choices between coarse-grained page sizes. This limitation of only supporting coarse-grained huge pages with a 512x factor between subsequent sizes presents too costly a tradeoff between memory bloat

to internal fragmentation and reduction in TLB entries required. When fragmentation is high, it is possible that insufficient contiguity exists to effectively use huge pages at all, whereas some benefit could be extracted by TPS+. ARM's additional page granule choices of 16KB/32MB or 64KB/512MB [4] only worsens the tradeoff. While these larger page granules do increase the minimum base page size, ARM's solution does not address the drawbacks the larger base page size introduces, nor does it leverage the microarchitectural benefits of the increased size. These limitations are evaluated throughout my dissertation.

Intel Itanium [32, 16] and SPARC [73] offer more page sizes, but these mechanisms require software controlled TLB structures to pin TLB entries to improve performance. TPS+ is largely orthogonal to these approaches since it eliminates most page walks altogether. Software controlled TLB approaches could still be used with TPS+ to accelerate page walks when they may be required. Itanium splits the address space into 8 regions, each with a configurable page size. This approach limits benefit despite the many page sizes offered.

HP Tunable Base Page Size [26] allows the minimum page size to be configured in the OS. This approach does not address the drawbacks introduced by the larger page size, hence why HP recommends using no larger than 16KB as the minimum size. The chosen minimum page size does not affect the actual microarchitectural base page size.

Romer et al.'s work [64] in superpages considered adding more variability to available page sizes, but this approach only evaluates a page relocation based approach to merge and promote smaller pages into larger pages. My frame reservation and eager allocation based approach reduces the need to perform extraneous memory copies to create large pages. In addition, this work only considers a software-managed TLB and does not de-

scribe the hardware changes necessary to support additional page sizes for hierarchical radix tree page tables.

Shadow superpages [74, 23] create a secondary translation mechanism maintained by the memory controller. Shadow memory space (i.e., unused physical addresses that do not correspond to installed physical memory) is divided into fixed sized superpages comprised of actual physical memory base pages that can be discontinuous (located anywhere in physical memory). The memory controller maintains its own special page table to translate from a shadow superpage to its component real physical pages. While this allows the core TLBs to achieve greater reach by caching PTEs for shadow memory superpages, translation is still required by the memory controller for all memory traffic, ultimately postponing page table memory references to later in the memory access process. This is particularly problematic for large memory workloads with high memory traffic. TPS+ nearly eliminates the translation latency even in cases of high memory intensity.

Ingens [49] is a purely operating system proposal that significantly improves on Transparent Huge Pages [18] by offering cleaner tradeoffs between memory consumption, performance, and latency. HawkEye [59] is another OS technique that further improves upon Ingens. HawkEye balances fairness in huge page allocation across multiple processes, performs asynchronous page pre-zeroing, de-duplicates zero-filled pages, and performs fine-grained page access tracking and measurement of address translation overheads through hardware performance counters. Because TPS and the increased base page size provide improvements to the underlying hardware, my mechanisms and techniques like Ingens/HawkEye could work cooperatively to improve these tradeoffs. TPS+ can additionally supply fine-grained metadata information about larger pages to the OS. By offering

more choice in page sizes, my work opens the door to further interesting OS research like Ingens/HawkEye along this path.

Mosaic [5] is a GPU-specific operating system memory manager that provides application-transparent support for multiple page sizes. This technique improves the balance between the benefits of larger pages and demand paging in GPUs. Mosaic specifically targets GPUs and only leverages existing page sizes; offering more options with TPS+ may enable further reduction in translation latency in GPU applications still suffering from significant numbers of TLB misses.

Various prior work [78, 80, 79, 71] have suggested the benefits in increasing the base page size, but they are all incomplete. [71] performs no evaluation, and none completely address how to appropriately deal with backwards compatibility and the additional latency incurred by the larger base page sizes. Additionally, none of these proposals deal with the problem of coarse-granularity in successive available page sizes.

5.3 Segmentation-Like Approaches

Early commercial processors have used segmentation for address translation. Several processors provided support for segmentation without paging [31, 33, 52]. Other processors supported both segmentation and paging [41]. Unlike past segmentation approaches, TPS+ adheres to the page-based virtual memory paradigm, enabling its benefits. TPS+ still allows for segmentation on top of paging.

Direct segment [8] is a segmentation-like approach to address translation. It is as an alternative to page based virtual memory for big memory applications that can utilize a single, large translation entity. A hardware segment maps one contiguous range of virtual address space to contiguous physical memory. The remaining virtual address space

is mapped to physical memory with the existing page-based virtual memory approach. A particular virtual addresses is translated to its physical address via either the hardware direct segment or the page table hardware and TLBs. Like standard segmentation, direct segment utilizes base, limit, and offset registers and does not require page walks within the segment. Unlike TPS+, this mechanism requires that the application explicitly creates a direct segment during its startup. The OS must at that time be able to reserve a single large contiguous range of physical memory for the segment. Direct segment requires application changes and is only suited for large memory workloads that can leverage a single, large segment.

Do it yourself virtual memory translation (DVMT) [1] is a technique that decouples address translation from access permissions and enables application code to construct custom mappings for a specific region of virtual memory. DVMT saves on translation latency for accesses to the custom mappings. The decoupled access permissions enables the OS to control access at a finer granularity (i.e., per physical frame) than the custom DVMT mapping. TPS+ does not require application code changes, and leaves virtual memory management to the operating system. TPS+ and DVMT could cooperatively exist in appropriate usage scenarios.

5.4 Sub-blocking TLBs

Sub-blocked TLBs [75], CoLT [62], and Clustered TLBs [61] combine a small number of nearby virtual pages that all map to nearby physical frames into a single TLB entry. These approaches rely on the default operating system behavior to assign small regions of contiguous or clustered physical frames to contiguous virtual pages. However,

these approaches are intrinsically limited to a small number (e.g., 16) of page translations per TLB entry.

Since a single TLB entry is used to translate multiple base pages, these mechanisms require storage for base page bit vectors in the TLBs. The bit vectors would continue to grow for larger numbers of pages within a single TLB entry. They rely on indexing a set-associative TLB with not the least significant bits of the virtual page number, but VPN shifted over by a factor corresponding to the sub-block factor (e.g., eight pages coalesced into one TLB entry shifts over three bits). CoLT showed that shifting more than three bits begins to lose performance. Moreover, they still require loading all individual page PTEs to verify mapping/translation validity for the particular sub-block. Because eight PTEs fit within a cache line, extra memory accesses are not required for coalescing up to eight PTEs. These limitations hinder the generality of the mechanisms' applicability to data sets of any size, thus limiting their potential benefits.

5.5 Reducing TLB Miss Cost

Various techniques to accelerate page walks seek to reduce the cost of TLB misses, rather than altogether reducing or eliminating the TLB misses. MMU caches reduce page walk latency by saving PTEs from higher page table levels, thereby bypassing at least one memory access during the page walk process [6, 11, 35]. Some current processors cache PTEs in the data cache hierarchy to reduce page walk latency on MMU cache misses [37]. These mechanisms can still be used with TPS+ to accelerate page walks when they are required. The larger base page improves MMU cache effectiveness. Creating appropriately tailored pages results in fewer total PTEs to represent the application address space, thereby using fewer cache lines for PTEs.

Inverted/Hashed Page Tables [30, 83] are alternatives to the hierarchical radix tree page table approach that reduce the number of memory accesses required to get the PTE upon a TLB miss. These techniques reduce page table memory overhead and require both hardware and software support. These mechanisms are harder to adapt to multiple larger page sizes and do not reduce the number of TLB misses, which TPS+ nearly eliminates.

The part-of-memory TLB (POM-TLB) [65] has proposed creating a large-capacity TLB in memory, with the ability to cache TLB entries in the chip cache hierarchy. This approach significantly reduces page walk latency by providing the opportunity to more quickly find the necessary TLB entry. In contrast, a primary benefit of TPS+ comes from the reduction in TLB misses resulting from the reduced total number of PTEs needed to translate the entire application's address space. When TLB misses do occur, TPS+ can work together with this approach to further minimize translation overhead.

5.6 Reducing TLB Miss Rate

Address translation overhead can be lowered by reducing the TLB miss rate. Synergistic TLBs [72] and shared last-level TLBs [53, 13, 10] seek to reduce the number of page walks and improve TLB reach.

Prior work has proposed hardware PTE prefetchers [14, 43, 67]. These approaches prefetch PTEs into the TLB before they are needed for translation. However, memory access pattern predictability limits TLB prefetcher effectiveness; for example, in applications with random access behavior, TLB prefetching will be unlikely to help. Other prior work has proposed speculative translation based on huge pages [7]. Like with TLB prefetching, this mechanism favors sequential patterns and relies on address contiguity.

In addition, [60] proposed a prediction technique that allows a single set associative TLB to be shared by all page sizes. Other prior work has proposed gap-tolerant mechanisms that allow coarse-grained superpage creation even when retired physical pages cause non-contiguity in available physical memory [22]. However, each TLB entry still only maps a single coarse-grained page. This limits TLB reach for memory intensive applications. Unlike these approaches, TPS+ creates translations for page sizes tailored to the application's data set, caching them in the TLB. TPS+ can work together with these approaches to improve translation latency.

5.7 Fine Grained Memory Protection

Prior work in fine-grained memory protection [25, 77, 81] identify similarity and contiguity across many coarse-grained base pages, similar to how TPS+ identifies contiguity in order to tailor a page of appropriate size. However, these approaches only leverage the contiguity of fine-grained protection rights across these larger ranges, while TPS+ leverages and further enhances the address space contiguity during memory allocation and reservation to facilitate faster address translation by greatly improving TLB hit rates.

5.8 Virtual Caching

Prior work in virtual caches seeks to reduce translation overhead by only translating after a cache miss [9, 82]. However, for workloads that suffer from poor locality which results in many TLB misses, virtual caches just shift the still-necessary translation to a higher level of the cache hierarchy while increasing system complexity in order to deal with the synonym problem. The translation penalty will still be incurred when physical addresses are actually needed, which TPS+ seeks to nearly eliminate.

Chapter 6

Conclusion

6.1 Summary

I have shown that both page walks and L1 TLB misses that hit in the L2 TLB result in significant performance penalties on current applications. To reduce translation latency from these sources, I have proposed TPS+. Additionally, TPS+ facilitates improved microarchitectural designs via the larger base page size. There are drawbacks to increasing the base page size, however. I have presented solutions to these drawbacks. TPS+ requires small changes to the microarchitecture, ISA, and operating system software. At very low memory overhead, TPS+ can eliminate more than 99% of all TLB misses and page walk memory references across a variety of SPEC17 and big data memory intensive benchmarks, yielding 59.7% average performance improvement in virtualized execution scenarios.

TPS+ is a promising mechanism for improving application performance and scalability as the growth in main memory capacity continues.

6.2 Limitations and Future Work

The work presented in this dissertation can certainly be further investigated and hopefully improved. I suggest the following:

- *Improved TLB Energy Efficiency*. Prior work [45] has shown that it is possible to turn off TLB ways that have low utilization and are unnecessary in the presence

of alternate translation mechanisms. Different applications have different levels of TLB utilization that require dynamic monitoring and adjustment based on runtime demand. It is likely possible to apply these concepts to TLBs utilizing TPS+. The total number of PTEs required with TPS+ is usually small, potentially allowing for significant energy reduction, particularly in the L2 TLB that may not require many active TLB entries. There are many details that must be addressed. With the conservative reservation and page promotion policy, TLB demand will begin high and slowly reduce as fewer PTEs are necessary to represent the application working set. With many ways potentially being disabled in the L2 TLB, how to tile the available page sizes amongst the TLB entries is particularly important.

- *Operating System Policies.* I have only implemented in simulation and evaluated the minimal OS features and policies necessary for base TPS+ functionality. Real OS implementation on hardware with TPS+ ISA and microarchitectural support is necessary for real-world adoption. In the presence of heavy fragmentation, intensive competing memory allocations from co-running applications, and significant interleaving of allocations and deallocations potentially resulting in many address space holes, TPS+ will be unlikely to achieve its ideal benefits. How to coordinate memory compaction, reservations, and access/dirty metadata bit vectors to maximize TPS+ benefit in these scenarios is an open problem.

- *Future Microarchitectural Enhancements.* The larger base page size allows for further exploration of microarchitectural design tradeoffs. I mentioned in Chapter 4 potential improvements to hardware prefetching due to less frequent page boundaries. SIPT [84] identified that even lower-latency, smaller L1 cache designs (i.e., 32 KB, 2-way set associative) are preferable to slightly larger, but longer latency,

L1 caches (i.e., 128KB, 4-way set associative). L2 caches have recently increased in size to 1MB [37]. There may be room to introduce another intermediate level to the cache hierarchy between these two sizes. What the right sizes, latencies, and inclusion/exclusion policies are for each level remains to be explored.

- *Virtual Machine Paging Policies.* On oversubscribed virtual machines, maintaining matching page sizes from guest virtual to host virtual to machine physical may be challenging. How to maximize host page sizes (to minimize translation latency) while balancing the costs of breaking pages, swapping, and live migration is an interesting research problem.

- *Beyond Machine Main Memory Translation.* Currently, virtual memory is limited to translation from application virtual address space to main memory physical address space. FlashMap [29] has shown there is significant benefit to be gained by merging across the layers: virtual memory translation, storage level translation (i.e., the file index/filesystem), and device level translation (i.e., the flash translation layer). Mellanox [56] have discussed data center wide virtual memory translation. How to adapt TPS+ to these scenarios is an open research question.

Bibliography

[1] H. Alam, T. Zhang, M. Erez, and Y. Etsion. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 457–468. ACM, 2017.

[2] AMD Corporation. *AMD64 Architecture Programmer's Manual*, March 2017.

[3] Apple Inc. *Apple Developer Guide: About the Virtual Memory System*, May 2013.

[4] ARM Holdings. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*, 2015.

[5] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. Mosaic: a gpu memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 136–150. ACM, 2017.

[6] T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 48–59, 2010.

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

[8] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 237–248. ACM, 2013.

[9] A. Basu, M. D. Hill, and M. M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, 2012.

[10] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee. Scalable distributed last-level tlbs using low-latency interconnects. In *Microarchitecture (MICRO), 2018 51st Annual IEEE/ACM International Symposium on*, 2018.

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

[12] A. Bhattacharjee. Advanced concepts on address translation. 2018.

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

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

[15] Z. Bodek. Transparent superpages for freebsd on arm. 2014.

[16] M. Chapman, I. Wienand, and G. Heiser. Itanium page tables and tlb. 2003.

[17] J. Corbet. /dev/ksm: dynamic memory sharing. `https://lwn.net/Articles/306704/`, November 2008.

[18] J. Corbet. Transparent hugepages. https://lwn.net/Articles/359158/, October 2009.

[19] J. Corbet. Huge page part 1 (introduction). https://lwn.net/Articles/374424/, February 2010.

[20] J. Corbet. Memory compaction. https://lwn.net/Articles/368869/, January 2010.

[21] J. Corbet. Split pmd locks. https://lwn.net/Articles/568076/, September 2013.

[22] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 223–234, 2015.

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

[24] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 178–189, 2014.

[25] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A case for unlimited watchpoints. In *Proceedings of the Seventeenth International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS XVII, pages 159–172, 2012.

[26] Hewlett-Packard. *Tunable Base Page Size*, 2008.

[27] S. Hily, Z. Zhang, and P. Hammarlund. Resolving false dependencies of speculative load instructions, Oct. 13 2009. US Patent 7,603,527.

[28] K. A. Hua, L. Liu, and J. Peir. Designing high-performance processors using real address prediction. *IEEE transactions on computers*, 42(9):1146–1151, 1993.

[29] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan. Unified address translation for memory-mapped ssds with flashmap. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 580–591. ACM, 2015.

[30] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 39–50. ACM, 1993.

[31] Intel 8086. `http://en.wikipedia.org/wiki/Intel_8086`.

[32] Intel® itanium® architecture developer's manual, vol. 2. `http://www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-software-developer-rev-2-3-vol-2-manual.html`.

[33] Intel Corporation. *Introduction to the iAPX 432 Architecture*, 1981.

[34] Intel Corporation. *80386 programmer's reference manual*, 1986.

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

[36] Intel Corporation. *5-Level Paging and 5-Level EPT*, May 2017.

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

[38] Intel optane technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html`, 2018.

[39] Intel skylake. `https://www.7-cpu.com/cpu/Skylake.html`.

[40] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[41] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, July 1998.

[42] B. Jacob and T. Mudge. Performance analysis of the memory management unit under scale-out workloads. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization*, pages 1–12, 2014.

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

[44] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal. Redundant memory mappings for fast access to large memories. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 66–78. ACM, 2015.

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

[46] A. Karp and R. Gupta. Sectored virtual memory management system and translation look-aside buffer (tlb) for the same, Aug. 31 1999. US Patent 5,946,716.

[47] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[48] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE micro*, 30(4):8–19, 2010.

[49] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with ingens. In *OSDI*, pages 705–721, 2016.

[50] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[51] L. Liu. History table for prediction of virtual address translation for cache access, Feb. 21 1995. US Patent 5,392,410.

[52] W. Lonergan and P. King. Design of the b 5000 system. *Datamation*, 7(5), May 1961.

[53] D. Lustig, A. Bhattacharjee, and M. Martonosi. Tlb improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level tlbs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1):2, 2013.

[54] G. Mathews. Method and apparatus for accessing a cache using index prediction, Sept. 21 1999. US Patent 5,956,752.

[55] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson. *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[56] Mellanox. Virtualizing data center memory for performance and efficiency. January 2009.

[57] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, pages 22–31, 2009.

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

[59] A. Panwar, S. Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360. ACM, 2019.

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

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

[62] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. Colt: Coalesced large-reach tlbs. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 258–269. IEEE, 2012.

[63] S. Phillips. M7: Next generation sparc. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–27. IEEE, 2014.

[64] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing tlb and memory overhead using online superpage promotion. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 176–187. ACM, 1995.

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

[66] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM.

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

[68] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, et al. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197. ACM, 2013.

[69] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. *ACM SIGARCH Computer Architecture News*, 43(3):79–91, 2016.

[70] A. Seznec. Concurrent support of multiple page sizes on a skewed associative tlb. *IEEE Transactions on Computers*, 53(7):924–927, 2004.

[71] R. L. Sites. Larger pages, January 2017.

[72] S. Srikantaiah and M. Kandemir. Synergistic tlbs for high performance address translation in chip multiprocessors. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 313–324, Dec. 2010.

[73] Sun Microsystems. *UltraSPARC T2 Supplement to the UltraSPARC Architecture*, 2007.

[74] M. Swanson, L. Stoller, and J. Carter. Increasing tlb reach using superpages backed by shadow memory. *ACM SIGARCH Computer Architecture News*, 26(3):204–213, 1998.

[75] M. Talluri and M. D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 171–182, 1994.

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

[77] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood. A small cache of large ranges: Hardware methods for efficiently searching, storing, and updating big dataflow tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 94–105, 2008.

[78] P. Weisberg and Y. Wiseman. Using 4kb page size for virtual memory is obsolete. In *Information Reuse & Integration, 2009. IRI'09. IEEE International Conference on*, pages 262–265. IEEE, 2009.

[79] P. Weisberg and Y. Wiseman. Virtual memory systems should use larger pages. *Advanced Science and Technology Letters*, 2015.

[80] P. Weisberg and Y. Wiseman. Virtual memory systems should use larger pages rather than the traditional 4kb pages. *memory*, 8(8), 2015.

[81] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, 2002.

[82] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. An in-cache address translation mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 358–365, 1986.

[83] I. Yaniv and D. Tsafrir. Hash, don't cache (the page table). In *ACM SIGMETRICS Performance Evaluation Review*, volume 44, pages 337–350. ACM, 2016.

[84] T. Zheng, H. Zhu, and M. Erez. Sipt: Speculatively indexed, physically tagged caches. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 118–130. IEEE, 2018.