Improving the Performance and Energy-efficiency of Virtual Memory

A Range-based Approach

Vasileios Karakostas

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

April 2016

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Acta de calificación de tesis doctoral	**Curso académico:**

Nombre y apellidos

Programa de doctorado

Unidad estructural responsable del programa

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada ____
____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a		(Nombre, apellidos y firma) Secretario/a
(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

______________________, _______ de __________________ de _______________

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretaria de la Comisión Permanente de la Escuela de Doctorado

Barcelona a _______ de ____________________ de __________

Dedicated to my family for always making the impossible possible

Abstract

Virtual memory improves programmer productivity, enhances process security, and increases memory utilization. These benefits are provided by introducing an indirection level between the virtual address space that the process sees and the physical memory that the operating system manages and allocates for each process. Thus, virtual memory requires an address translation from the virtual to the physical address space on every memory operation.

Page-based implementations of virtual memory divide physical memory into fixed size pages, and use a per-process page table to map virtual pages to physical pages. The hardware key component for accelerating the address translation is the Translation Lookaside Buffer (TLB), that holds recently used mappings from the virtual to the physical address space. The TLB used to be a small monolithic structure. Due to the criticality of the TLB in the system's performance, commodity processors have employed a per-core two-level TLB organization with additional support for huge pages. However, the address translation still incurs high (i) performance overheads due to costly page table walks after TLB misses, and (ii) energy overheads due to frequent TLB lookups on every memory operation. This thesis quantifies these overheads and proposes techniques to mitigate them.

In this thesis we argue that fixed size page-based approaches for address translation exhibit limited potential for improving TLB performance because they increase the TLB reach by a *fixed amount*. To overcome the limitations of such approaches, we introduce the concept of *range translations* and we show how they can significantly improve the performance and energy-efficiency of address translation.

We first comprehensively quantify the address translation performance overhead on a collection of emerging scale-out applications. We show that address translation accounts for up to 16% of the total execution time. We find that huge pages may improve the application performance by reducing the time spent in page walks, enabling better exploitation of the available execution resources. However, the limited hardware support for huge pages in combination with the workloads' low memory locality leave ample space for performance optimizations. In response, we present upper bounds for perfect optimizations in the address translation path that motivate rethinking its design in the context of memory intensive applications.

To reduce the performance overheads of address translation, we propose *Redundant Memory Mappings (RMM)*. RMM leverages *ranges* of pages and provides an efficient alternative representation of many virtual-to-physical mappings. We define a *range translation* be a subset of a process's pages that are virtually and physically contiguous. RMM translates each range translation with a single *range table entry*, enabling a modest number of entries to translate most of the process's address space. RMM operates in parallel with standard paging and introduces a software range table and a hardware range TLB with arbitrarily large reach that is accessed in parallel with the regular L2-page TLB. We modify the operating system to automatically detect ranges and to increase their likelihood with *eager page allocation*. RMM is thus transparent to applications. We prototype RMM software in Linux and emulate the hardware. RMM reduces the overhead of virtual memory to less than 1% on average on a wide range of workloads.

To reduce the energy cost of address translation, we propose the *Lite* mechanism and the TLB$_{Lite}$ and *RMM*$_{Lite}$ designs. Lite is a mechanism that monitors the performance and utility of L1 TLBs, and adaptively changes their sizes with way-disabling. The resulting TLB$_{Lite}$ organization targets commodity processors with TLB support for huge pages and opportunistically reduces the dynamic energy spent in address translation with minimal impact on TLB miss cycles. To further provide more energy-efficient address translation, we propose RMM$_{Lite}$ that leverages the RMM address translation mechanism. RMM$_{Lite}$ adds to RMM an L1-range TLB, that is accessed in parallel with the regular L1-page TLB, and the Lite mechanism. The high hit ratio of the L1-range TLB allows Lite to downsize the L1-page TLBs more aggressively. RMM$_{Lite}$ reduces the dynamic energy spent in address translation by 71% on average. Above the near-zero L2 TLB misses from RMM, RMM$_{Lite}$ further reduces the overhead from L1 TLB misses by 99%.

The proposed designs target current and future high-performance and energy-efficient memory systems to meet the ever increasing memory demands of applications.

Acknowledgements

My “short” journey in Barcelona started with a six-month visit as part of my undergraduate studies and ends after several years with completing a PhD degree (for the moment). The journey was not short. But it was a really great experience, thanks to the support of many people that I wish to acknowledge here.

First of all, I would like to express my deepest gratitude to my advisors Mario Nemirovsky, Osman S. Unsal, and Adrian Cristal. Mario’s enthusiasm and solid criticism in discussing ideas helped me a lot in identifying interesting problems and asking the right questions. His extremely positive and supportive character helped me in surpassing the obstacles that I encountered in my way. Osman and Adrian trusted me from day one, first with accepting me as an intern, and later with allowing me to continue my graduate studies at Barcelona Supercomputing Center. I sincerely thank them for their sound and technical advice, but more importantly for their immense confidence, endless help, and constant support, and for giving me the freedom to work on what I was interested in.

I also had the unique opportunity to have three exceptional external faculty collaborators that I would like to gratefully thank: Prof. Mark D. Hill, Prof. Michael M. Swift, and Prof. Kathryn McKinley. They all greatly helped me in shaping and communicating the ideas presented in this thesis. Moreover, they taught me many things about research, and contributed further into my transition from student to researcher. In particular, I feel indebted to Michael for the many technical discussions we had during his sabbatical in Barcelona, and for forming our great collaboration team. In addition, I would like to especially thank Mark, Michael, Prof. David A. Wood, and all the people from the Wisconsin Multifacet Project for making me feel more than welcomed in Madison during my short visit—an experience I will always remember.

I would like to thank Jayneel Gandhi for having the great pleasure to meet and collaborate extensively. His hard-working but also positive attitude helped me a lot with meeting difficult deadlines. I would also like to thank Jayneel for his help in this thesis, especially in the RMM work.

I would like to thank Prof. Guri Sohi, Prof. Uri Weiser, and Prof. Francisco Cazorla for serving in my thesis committee and providing very useful suggestions that improved this document.

I would like to thank all my collaborators and co-authors, and particularly Gokcen Kestor, Ibrahim Hur, and Carlos Villavieja for their help and support at the beginning of my graduate studies.

During my graduate studies I had the luck to meet and exchange ideas with many great researchers. In particular, I would like to thank Prof. Avi Mendelson and Dr. Tim Harris for always providing constructive feedback.

I would like to thank Xavier Salazar for his help and patience in preparing the documents for my scholarship applications.

I am also indebted to all my friends and colleagues from Barcelona Supercomputing Center for their support during these years. Many thanks go to Nehir Sonmez, Adria Armejach, Sasa Tomic, Vladimir Subotic, Oriol Arcas, Srdjan Stipic, Gulay Yalcin, Vesna Smiljkovic, Azam Seyedi, Nikola Markovic, Daniel Nemirovsky, Ferad Zyulkyarov, Behzad Salami, Furkan Ayar, Damian Roca, Francesco Ciaccia, Josue Quiroga, Vladimir Gajinov, Oscar Palomar, Omer Subasi, Javier Arias, Timothy Hayes, Milovan Djuric, Milan Stanic, Ivan Ratkovic, Cristian Perfumo, Paul Carpenter, Nebojsa Miletic, and Ruben Titos. I sincerely thank them all for their help and all the great moments we had together.

Finally, I would like to thank my friends and family for supporting me during all these years. My deepest thanks to Efi for her love, for being next to me, and for reminding me always that there is also a life to live. This dissertation would have not been possible without her.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In this chapter, we first present the basic concepts of page-based virtual memory and the evolution of the address translation support. We then discuss the motivation behind our work and state the problems that we tackle in this thesis. Then we present our approach for reducing the performance and energy overheads of virtual memory, and, finally, we provide an overview of our contributions.

1.1 Virtual Memory

Virtual memory provides the programmer with an “infinite” amount of memory by virtualizing the available physical memory. This functionality increases programmer productivity, enables process isolation, and enhances system consolidation.

To provide all these benefits, virtual memory introduces an indirection level between the virtual address space that each process sees and the physical memory that the operating system manages and allocates for all processes. However, the indirection cost of virtual memory does not come for free: *virtual memory implies that each memory operation made* *by an application requires an address translation to obtain the physical address, incurring high performance* [27, 28, 68, 77, 90] *and energy* [2, 3, 49, 71, 72, 73] *overheads*.

1.1.1 Architectural Support

Page-based implementations of virtual memory are ubiquitous in modern computing systems. They divide physical memory into fixed size pages, typically 4 KB, and use a per-process *page table* to map virtual pages to physical pages. The page table is typically organized in an hierarchical fashion that requires multiple memory references for retrieving a virtual to physical mapping. The Memory Management Unit (MMU) is responsible for performing fast address translation in hardware, avoiding accesses in the page table. The MMU primarily consists of the Translation Lookaside Buffer.

Translation Lookaside Buffer (TLB) is the hardware key component for accelerating address translation. The TLB holds recently used mappings from the virtual to the physical address space. The processor accesses the TLB on every memory request to obtain the address translation, either before or in parallel with accessing the cache hierarchy. In case of TLB hit, the address translation is obtained fast, and the memory request continues with accessing the memory hierarchy. However, in case of TLB miss, a page walk occurs, i.e., a hardware state machine walks the page table. The page walk introduces extra multiple memory references to fetch the address translation from the page table, e.g., four memory references in the x86-64 architecture. Thus, the performance of the TLB depends on the *TLB reach*, i.e., the memory for which the TLB may service address translation requests without experiencing a miss.

1.1.2 Evolution of Address Translation Hardware Support

For a long time since their invention in the 1960s [44], TLBs have been a small monolithic structure and were able to deliver high performance. Commercial processors, however, keep on devoting more resources to memory and address translation to meet the ever increasing memory demands of memory intensive workloads. The common organization of the MMU found in today's processors includes: (i) per-core multi-level TLBs, (ii) with support for huge pages, (iii) backed by MMU caches.

Two-level TLB organization. Due to the criticality of the TLB in the system's performance, processor vendors have employed a per-core two-level TLB organization. The L1 TLB is small and features a very fast search operation to serve the processor as fast as possible. The L2 TLB is larger and slower than the L1 TLB, and aims at holding as many translations as possible in order to reduce the number of the costly page walks. To boost the system's performance further, processors provide separate TLBs for data and instruction accesses.

TLB Support for Huge Pages. To increase further the TLB reach and improve TLB performance, processors provide TLB support for huge pages. For example, x86-64 architectures provide TLB support for 2 MB and 1 GB pages, in addition to 4 KB pages. Thus, a single TLB entry for a huge page maps the same memory as if multiple TLB entries for regular base pages were used.

MMU cache. To reduce the impact of the page walk latency in performance, commercial processors back the TLB organization with MMU caches. The MMU cache reduces the cost of page walks by caching intermediate levels of the page table, while the TLB only caches the leaves of the page table. A hit in the MMU cache enables the processor to skip one or more levels of the page table in order to complete the address translation with less memory operations.

1.2 Motivation

Mismatch Between TLBs, Memories, and Emerging Workloads

Page-based virtual memory used to deliver high performance, since TLBs serviced the vast majority of address translation requests. Unfortunately, the performance of paging is suffering. While memory sizes—both on-chip and off-chip—increase due to Moore's law, TLB sizes have merely increased within the evolution of processors with respect to memories. The reason for this mismatch is straight-forward: *TLBs are on the critical path of accessing the memory hierarchy*. Adding more entries in the TLBs may increase the hit ratio. However, the TLB latency also increases due to the larger size, affecting the translation cost for all memory operations—including TLB hits. In addition, because the TLBs are accessed on every memory reference, they consume important percentage of processor energy. Adding more entries in the TLBs will also increase their energy consumption. Consequently, the

TLBs remain practically stagnant with respect to memory sizes, and introduce significant performance and energy overheads for those workloads that deal with large working sets and exhibit poor memory locality. Without new designs, the mismatch between TLBs, memory sizes, and application behavior will likely keep growing.

1.3 Problem Statement

Previous works have highlighted the impact of the address translation path in terms of performance and energy overheads. This thesis quantifies these overheads and proposes techniques to mitigate them.

High Address Translation Performance Overheads. The performance of page-based virtual memory is suffering due to *limited TLB reach*. Recent studies and this thesis show that modern workloads can experience performance overhead due to page table walks [27, 28, 68, 77, 90]. This overhead is likely to grow, because physical memory sizes are still growing.

The first goal of this thesis is to quantify the performance overhead of address translation for an emerging class of workloads, and to eliminate the performance overheads of virtual memory with a robust virtual memory implementation that is transparent to applications and that enables fast address translation across a variety of workloads.

High Address Translation Energy Overheads. The TLBs have been reported to consume a significant fraction of energy spent by the processor since the time they were monolithic [2, 3, 49, 71, 72, 73]. The recent growth in the complexity of the TLBs has further increased their energy consumption. A recent industrial report suggests that TLBs are responsible for 3-13% of a processor's power [108].

The second goal of this thesis is to analyze the sources of inefficiency in the address translation path, and to improve opportunistically the energy efficiency of TLBs in the presence of mechanisms that increase TLB reach while improving also the performance.

1.4 Thesis Approach

Fixed size page-based approaches exhibit limited potential for improving TLB performance because they increase the TLB reach by a *fixed amount*. As memory sizes increase more aggressively than TLB sizes, we believe that the virtual memory overheads that manifest in today's systems with 4 KB pages, will manifest similarly in tomorrow's systems with larger but fixed size mappings. Our experiments show that such cases exist already.

In this thesis we take a different approach and introduce the concept of *range translations* for translating contiguous virtual pages that are mapped to contiguous physical pages. Range translations enable an efficient alternative representation of many virtual-to-physical mappings. We show that range translations can significantly reduce the performance and energy overheads spent in address translation. Hence, we argue that range translations is the next logical step in the evolution of virtual memory.

1.5 Thesis Contributions

This thesis makes the following contributions:

- We quantify the performance overheads of address translation under the execution of scale-out applications that dominate in datacenter computing. We find that a significant percentage of the total execution time is spent in page walks due to TLB misses, even when huge pages are employed.

- To reduce the address translation performance overheads, we propose *Redundant Memory Mappings* (RMM), a hardware/software co-designed implementation of virtual memory that reduces significantly the number of TLB misses that trigger page walks through the notion of range translations.

- To reduce the energy overheads and to provide energy-efficient address translation, we introduce the *Lite* mechanism, and we propose TLB_{Lite} that targets commodity processors with TLB support for huge pages, and RMM_{Lite} that builds on RMM and leverages the architectural support for ranges.

Next we highlight the most important concepts of each contribution.

1.5.1 Quantifying Address Translation Overheads

Scale-out applications target various domains of datacenter computing including data analytics, key-value caching and storing, graph analytics, and web-searching, among others. These applications operate on large datasets with low memory locality, exhibiting inefficient execution in traditional server architectures [50]. In response, researchers proposed novel designs to increase the efficiency of various components of the microprocessors for scale-out applications [69, 86, 87]. However, the performance overhead of address translation in the Memory Management Unit (MMU) for scale-out applications has been largely ignored. There have been very few studies that primarily focused on solutions to mitigate the performance cost of the MMU [27, 28].

In this thesis, we perform a comprehensive performance analysis of the MMU under the execution of various scale-out applications. We conduct our analysis leveraging the use of performance counters on an x86-64 real system.

We find that the performance overhead of address translation accounts for up to 16% of the total execution time, due to the high number of TLB misses that trigger page walks and the interference between page walks and application data in the cache hierarchy. We find that the performance improves by reducing the time spent in page walks, enabling better exploitation of the available execution resources.

We observe that huge pages are beneficial for most applications without being an "always-win" option due to limited hardware support. We also quantify the interference between the application data and the page table in the cache hierarchy, and show how page walks are affected by hardware prefetchers.

Finally, we present upper-bound analyses and provide potential directions for improving the MMU performance.

1.5.2 Reducing Address Translation Performance Overheads

Page-based virtual memory incurs high performance overheads due to costly page table walks after TLB misses. Previous research has aimed on increasing TLB reach by improving the efficiency of paging with: (i) Multipage mappings [96, 97, 111], that translate several pages with a single TLB entry, (ii) Huge pages [5, 8], that translate much larger aligned memory with a single TLB entry, and (iii) Direct segments [27, 52], that provide a single arbitrarily large segment along with standard paging. However, all these efforts suffer from various limitations; multipage mappings and huge pages have size and alignment restrictions, and still provide limited TLB reach with respect to the ever-growing physical memories, while direct segments require application modifications and do not provide performance benefits for all workloads.

In this thesis, we propose *Redundant Memory Mappings* (RMM), a novel hardware/software co-designed implementation of virtual memory. RMM exploits the natural contiguity in address space and introduces a redundant mapping named *range translation*, in addition to page tables, that provides a more efficient representation of translation information for *ranges* of pages that are both virtually and physically contiguous with uniform protection. With range translations, RMM increases the TLB reach and reduces significantly the number of page walks, enabling a robust virtual memory implementation with near zero performance overhead.

RMM relies on the concept of *range translation*. Each range translation maps a contiguous virtual address range to contiguous physical pages with uniform protection access rights, and uses BASE, LIMIT, and OFFSET values to perform translation of an arbitrary sized range. Range translations are only base-page-aligned and redundant to paging; the page table still maps the entire virtual address space.

Analogous to paging, RMM introduces three novel components to perform address translation with range translations: (i) *range TLB*s, (ii) *range tables*, and (iii) *eager paging* allocation. More specifically, RMM introduces a hardware *range TLB* that is accessed in parallel with the L2-page TLB. The range TLB caches recently used range translations, accelerates their address translation, increases TLB reach, and reduces the number of page walks. RMM introduces also a software managed *range table* that stores in memory all range translations for each process. To increase contiguity in range translations, we extend the operating system's default lazy demand page allocation strategy to perform *eager paging*. Eager paging instantiates pages in physical memory at allocation request time, rather than at first-access time as with demand paging. Because range tables are redundant to page tables, RMM offers all the flexibility of paging and the operating system may use or revert solely to paging when necessary. The resulting operating system automatically maps most of process's virtual address space with orders of magnitude fewer ranges than paging.

Overall, RMM reduces the overhead of virtual memory to less than 1% on average, while combining the benefits and surpassing the limitations of previous proposals.

1.5.3 Improving Address Translation Energy-Efficiency

Page-based virtual memory incurs also high energy overheads because the TLB is accessed on every memory operation. Prior research has focused on reducing the dynamic energy of TLBs through various techniques [20, 21, 38, 38, 41, 49, 71, 82]. However, those energy optimization techniques do not take into account hardware support for increasing the TLB reach (e.g., huge pages).

In this thesis, we analyze the energy spent in the address translation path, using as baseline a common per-core two-level TLB organization with a separate set-associative L1 TLB for each supported page size, e.g., for 4 KB, 2 MB, and 1 GB pages. Our findings show that the L1 TLBs are the primary source of dynamic energy overhead in the address translation path. We also find that page walks consume significant amount of energy with 4 KB pages. While huge pages and other techniques that increase TLB reach [27, 51, 78, 96, 97, 111] reduce the energy due to page walks, we observe that the "innocent" L1 TLB hits remain the dominant source of dynamic address translation, because *multiple separate L1 TLBs are accessed on every memory operation*.

To reduce the energy cost of address translation, we first propose *Lite*. Lite monitors the utility of ways in the L1 TLBs for each page size in an interval fashion based on the distance of TLB hits from the least-recently-used (LRU) position, similar to the accounting cache [47] and utility-based cache partitioning [102]. At the end of each interval, Lite evaluates the utility of L1 TLBs. In case the utility of active ways is insignificant, Lite opportunistically downsizes each of the L1 TLBs individually by disabling ways [16]. Lite thus accesses fewer ways in the L1 TLBs, saving energy at the cost of introducing a few additional misses. The resulting TLB$_{Lite}$ organization targets commodity processors with TLB support for huge pages, requires minimal modifications, and opportunistically reduces L1 TLB energy with negligible impact on performance.

We additionally propose RMM$_{Lite}$ to further augment the potential of Lite for reducing the energy in L1 TLBs while at the same time reducing both the energy and performance overheads due to L1 TLB misses. RMM$_{Lite}$ builds on RMM, and introduces a small L1-range TLB and the Lite resizing mechanism. The L1-range TLB is accessed in parallel with the L1-page TLB and is small (e.g., 4 entries) in order to meet the tight timing requirements of L1-TLBs. Yet the L1-range TLB is powerful; each range TLB entry can hold a mapping of unlimited size, that enables the L1-range TLB to enjoys a high hit ratio. That allows Lite to downsize L1-page TLBs more aggressively without affecting the performance.

Our evaluation results show that TLB$_{Lite}$ reduces opportunistically the dynamic energy spent in address translation by 23% while slightly increasing the cycles spent in TLB misses compared to huge pages [5]. RMM$_{Lite}$ reduces the dynamic energy spent in address translation by 71% on average compared to huge pages. Above the near-zero L2 TLB misses from RMM, RMM$_{Lite}$ further reduces the overhead from L1 TLB misses by 99%.

Overall, TLB$_{Lite}$ and RMM$_{Lite}$ improve both energy efficiency and performance of address translation.

1.6 Thesis Organization

Chapter 2 provides additional background on virtual memory and address translation with emphasis on page-based systems with hardware-managed TLBs.

Chapter 3 analyzes the performance of the Memory Management Unit under scale-out workloads, focusing on the cost of TLB misses that trigger page walks and their interaction with other processor components. This chapter follows mostly from our work published in 2014 IEEE International Symposium on Workload Characterization (IISWC 2014) [77].

Chapter 4 presents Redundant Memory Mappings (RMM), a hardware/software co-designed implementation of virtual memory that eliminates the number of page walks. This chapter follows mostly from our work published in the 42nd International Symposium on Computer Architecture (ISCA 2015) [78] and summarized in the IEEE Micro Special Issue on Top Picks from 2015 Computer Architecture Conferences [53].

Chapter 5 presents the Lite mechanism, and the TLB$_{Lite}$ and RMM$_{Lite}$ organizations that improve the energy-efficiency of address translation. This chapter follows mostly from our work published in the 22nd International Symposium on High Performance Computer Architecture (HPCA 2016) [79].

Chapter 6 concludes this thesis and points to future research directions.

2 Background on Virtual Memory

This chapter provides background information on virtual memory. More specifically, we introduce the basic concepts of page-based virtual memory and the role of address translation, then we describe the software and hardware components of the architectural support for virtual memory, and finally we discuss briefly the variation of segment-based virtual memory. Since in this thesis we mainly focus in the x86-64 architecture, we explain here in more detail how address translation is performed in that architecture. Note that the problems that we tackle in this thesis are not specific to the implementation of virtual memory in the x86-64 architecture, i.e., high performance and energy overheads due to address translation, and that similar issues hold for other architectures as well. For more information about the implementation of virtual memory in other architectures, we refer the interested readers to [66, 67].

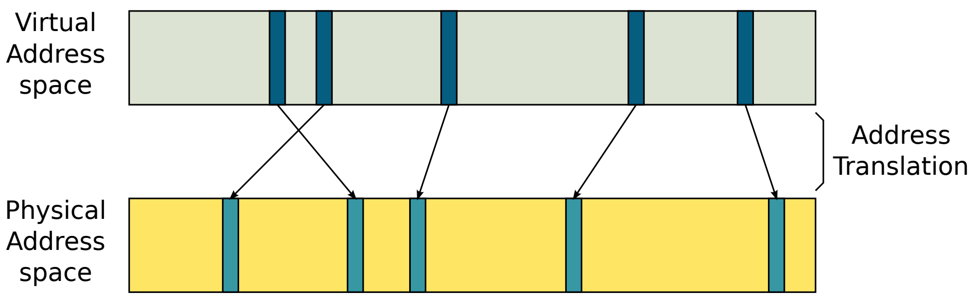


Figure 2.1: The abstraction of the page-based virtual memory. The virtual and the physical address spaces are divided into pages. The application "sees" the virtual address space. The operating system manages the physical address space. The address translation implements the abstraction of virtual memory with mapping the virtual addresses to physical addresses.

2.1 Virtual Memory

Virtual memory was originally introduced in the computing systems to overcome the problem of limited physical memory. Without virtual memory, the programmer had to make sure that the program fitted in the physical memory. This restriction was responsible for decreased software productivity and limited software portability. With virtual memory, the programmer sees a big flat memory without bothering about the actual implementation and limitations of the physical memory or memory system.

Virtual memory provides several benefits that have rendered its presence ubiquitous in the computing systems for several decades now. Among others, virtual memory improves process isolation and security and enhances programmer productivity, since the operating system manages the mappings from the per-process virtual address space to the system's physical address space. In addition, virtual memory enables the operating system to consolidate more efficiently multiple running processes by managing better the available physical memory.

2.2 Basic Concepts

The abstraction of page-based virtual memory relies on four basic concepts, as shown in Figure 2.1: the *virtual address space*, the *physical address space*, the *pages*, and the *address translation*. Next we introduce these basic concepts.

Virtual address space. The virtual address space is a set of address areas that the process sees. The operating system allocates these virtual address areas and makes them available to the process.

Physical address space. The physical address space is the actual physical memory, i.e., the main memory that a computing system is equipped with. The operating system manages the physical memory and allocates portions of the available physical memory to map parts or the entire virtual address space of a process.

Pages. Both virtual and physical address spaces are divided and managed in fixed size pages or in page granularity. The typical base size of pages for most architectures is 4 KB. The virtual address space is divided into uniform virtual pages, each of which is identified by a virtual page number. Similarly, the physical address space or physical memory is divided into uniform physical pages, each of which is identified by a physical page number or physical frame number.

Address Translation. When a process requests to access a memory location, an address translation from the virtual address space to the physical address space needs to be performed. The address translation is the function that provides that virtual-to-physical mapping; it receives as input a virtual page number, or simply a virtual address, and produces as output a physical page number, or simply a physical address. Based on the physical address, the process accesses the requested memory location and reads or writes data. Because the process always "sees" the virtual memory only, an address translation is necessary on every memory reference.

2.3 Architectural Support

The implementation of virtual memory is a great example of hardware/software co-design between the processor (hardware) and the operating system (software). The operating system allocates physical memory, and maps the virtual addresses of a process to physical addresses by keeping the translation information in a software structure called *Page Table*. The processor accelerates virtual memory with the *Memory Management Unit (MMU)*. The MMU consists of two kinds of special address translation caches: the *Translation Lookaside Buffer (TLB)* that holds recently used page table entries, and the *MMU cache* that holds

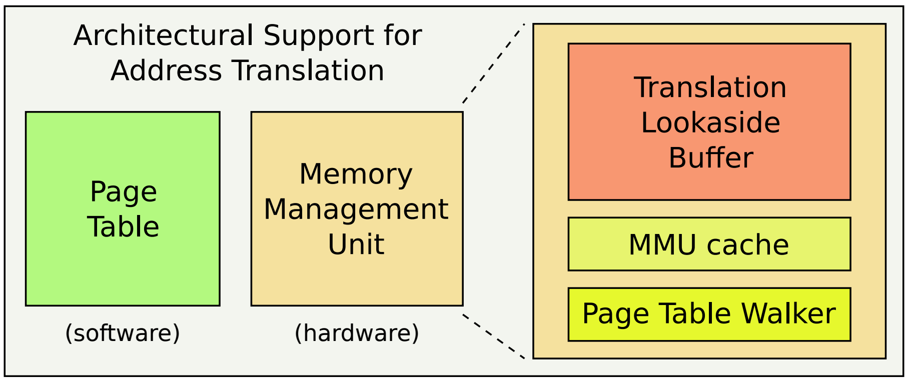


Figure 2.2: The architectural support for address translation consists of the Page Table (software component) and the Memory Management Unit (hardware component). The MMU consists of the Translation Lookaside Buffer, the MMU cache, and the page table walker.

intermediate levels of the page table. Figure 2.2 shows the basic components of the architectural support for address translation.

2.3.1 Page Table

The page table implements the abstraction between the virtual and the physical address space. The page table is an architecture-visible software data structure that is managed by the operating system. The page table stores in memory all the translations from the virtual to the physical address space for each process.

Page Table Entry (PTE). The page table consists of the page table entries. Each page table entry contains information for a virtual page that is (potentially) mapped by a physical page. This information is kept in a compact way, in order to keep the size of the page table entry reasonable and to prevent the page table itself from occupying the physical memory. The most common information found inside the page table entry across various architectures usually includes:

- the *valid bit* that indicates whether that page is actually mapped in the physical address space, and thus this page table entry holds valid information,
- the *physical page number* that holds the actual address translation of the corresponding virtual page,

- the *protection access rights* that define what memory operations are permitted in that page, e.g., no access, read-only access, or read/write access is permitted,
- the *privilege access level* that indicates whether user code or supervisor code can access that page,
- the *no-execute bit* that indicates whether that page may hold any code information or not, to prevent malicious software from inserting code into a data section and running their own code,
- the *cache-disabled bit* that defines whether that page is cacheable in the memory hierarchy or not,
- the *access bit* that indicates whether that page was recently accessed; the operating system uses this information to decide which pages to reclaim from the processes when serving new memory allocation requests, and
- the *modify bit* that indicates whether the processor wrote any data in that page; when set, the operating system writes back the contents of that page to the disk before reclaiming it from a process (i.e., swapping).

Hierarchical Page Table Organization. The page table could be organized as a flat table that holds all the mappings from the virtual to the physical address space for every process, including those mappings that correspond to currently non-allocated pages. However, such an organization would be clearly inefficient because it would waste a lot of memory for the page table itself.

To overcome the limitations of a flat table, the page table is usually organized in an hierarchical fashion[1]. The hierarchical page table organization splits the page table in various levels, so that entries in higher levels (closer to the root of the page table) hold information for larger regions of memory. Thus, the size of the hierarchical page table depends on the number of the virtual pages that a process uses.

Figure 2.3 shows the implementation of the page table for the x86-64 architecture. The page table is implemented in four levels named as PML4, PDP, PD, and PT. Figure 2.4 shows

[1]Another approach for organizing the page table is the inverted page table that is used in the PowerPC architecture [66, 67].

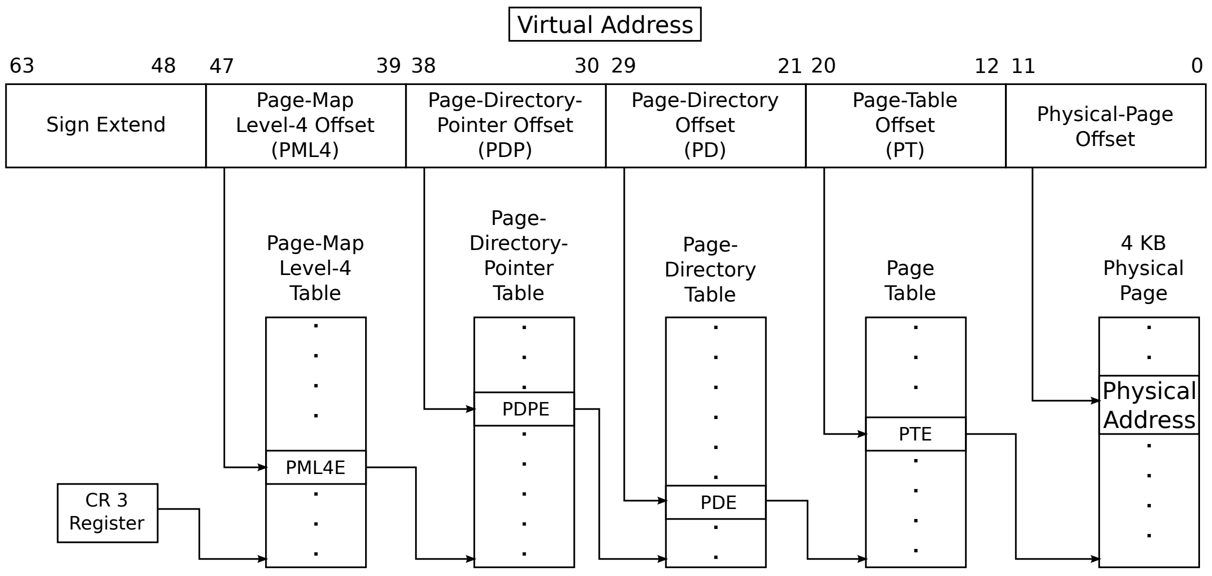


Figure 2.3: The design of the page table in x86-64 architecture. The page table is organized in a four-level hierarchical fashion, and thus address translation requires four memory operations.

the fields for each level of the page table. The virtual address is divided into parts and each part serves as index in that level's page table. The entries of each level hold pointers to the entries of the next level. The last-level PT of the page table (PTE) holds the translation information for that virtual address. Hence, each PT entry holds translation information for a single 4KB page of physical memory, each PD entry points to a PT and may hold translation information for a 2 MB memory area, each PDP entry points to a PD and may hold translation information for a 1 GB memory area, and each PML4 entry points to a PDP and may hold translation information for a 512 GB memory area.

Page Walk. Page walk is the process that involves: (i) accessing the page table with the virtual address, (ii) traversing the page table, and (iii) obtaining eventually the corresponding page table entry that contains the address translation for that virtual address. The x86-64 architecture walks the hierarchical page table in a *top-down* fashion[2]. The walk is performed by the page table walker, a hardware finite state machine as explained next.

The physical address of the root of the page table is stored in the *process control block*, a per-process data structure that the operating system uses to hold various information

[2]Another approach for accessing the hierarchical page table is the bottom-up method that is used in MIPS and Alpha architectures [66, 67].

63	62–59	58–52	51–M	M-1–30	29–21	20–13	12	11–9	8	7	6	5	4	3	2	1	0	bits
Reserved				Address of PML4 table				Ignored					PCD	PWT	Ign.			CR3
XD	Ignored		Rsvd.	Address of page-directory-pointer table				Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored																	0	PML4E: not present
XD	Prot. Key	Ignored	Rsvd.	Address of 1GB page frame	Reserved		PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored		Rsvd.	Address of page directory				Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored																	0	PDTPE: not present
XD	Prot. Key	Ignored	Rsvd.	Address of 2MB page frame		Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored		Rsvd.	Address of page table				Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																	0	PDE: not present
XD	Prot. Key	Ignored	Rsvd.	Address of 4KB page frame				Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																	0	PTE: not present

Figure 2.4: Format of the CR3 register and of the page table's entries of each level (PML4, PDP, PD, PT) for the x86-64 architecture [64]. M: the size of the physical address space, e.g., 48- to 52-bits. R/W: read/write bit. U/S: user/supervisor bit. PWT: page write through bit. PCD: page cache disable bit. A: accessed bit. D: dirty bit. G: global bit. XD: execute-disable bit.

for each process in the system. When the operating system schedules a process to a core (context switch), the OS copies also the physical address of the root of the page table in the CR3 register, an architecture-visible register. The hardware state machine uses the CR3 register to walk the page table when TLB misses occur.

Figure 2.3 shows the process of page walk for the x86-64 architecture, i.e., translating a virtual address to a physical address with the page table. The CR3 register points to the root of the PML4 table. The hardware walker uses the CR3 register as base, that points to the root of the PML4 table, and the 47-39 bits of the virtual address as index in the PML4 table to read the corresponding PML4 entry (first memory operation). That PML4 entry points to the root of a PDP table; the hardware walker uses the 38-30 bits of the virtual address as index in the PDP table to read the corresponding PDP entry (second memory operation). That PDP entry points to the root of a PD table; the hardware walker uses the 29-21 bits of the virtual address as index in the PD table to read the corresponding PD entry (third memory operation). Finally, that PD entry points to the root of a PT table; the hardware walker uses the 20-12 bits of the virtual address as index in the PT table to read the corresponding PT entry (fourth memory operation).

Page Fault. A page fault occurs when no page table entry or no valid translation is found in the page table, or when the process performs a memory operation that violates the access rights (e.g., read/write, user/supervisor, etc.) of that page. The page fault triggers an interrupt and the control is passed to the operating system that executes the page fault handler. The page fault handler is a software routine that first checks the reason that triggered that page fault, and then takes the corresponding action. In case the page fault was due to accessing for the first time a page that is allocated in the virtual address space but is not allocated yet in the physical address space (due to *demand paging*), then the operating system allocates a physical page and updates the corresponding page table entry. In case the page fault was due to accessing the backing storage (e.g., disk), then the operating system fetches the corresponding data from the backing storage into memory and updates the page table. Finally, in case the page fault was due to illegal access because of protection rights violation, then the operating system terminates the process with an error signal.

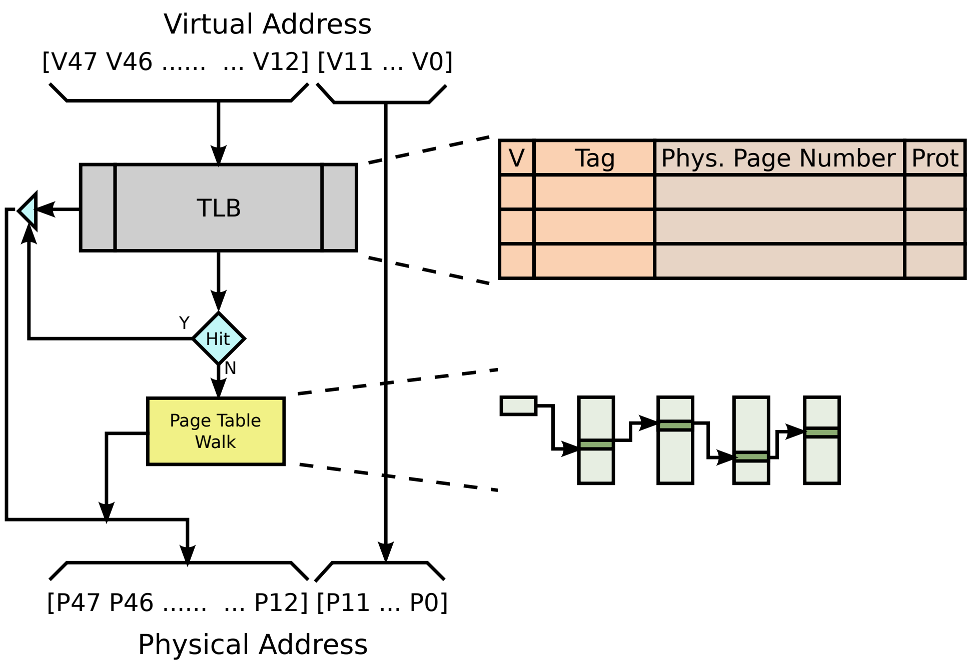


Figure 2.5: The Translation Lookaside Buffer accelerates address translation by caching recently used page table entries. In case of TLB miss, a hardware state machine walks the page table.

2.3.2 Translation Lookaside Buffer

With virtual memory, each memory operation made by a process requires an address translation to obtain the physical address. This would imply that for each memory operation in the x86-64 architecture, the processor should: (i) first perform four additional memory operations to obtain the physical address from the page table, and (ii) then perform the actual memory operation that was requested in the first place. Clearly such an approach would have high impact in the performance that would probably outweigh the benefits of virtual memory.

To accelerate virtual memory, processors employ the *Translation Lookaside Buffer (TLB)*. The TLB is a small cache that only holds recently used page table entries. With respect to Figure 2.3, the TLB holds mappings from virtual pages to physical pages bypassing the intermediate levels of the page table. The processor accesses the TLB on every memory operation with the requested virtual address. In case of a hit, the TLB returns to the processor the corresponding physical address, and the processor proceeds with accessing the memory hierarchy. Thus, a TLB hit provides fast address translation without accessing the page table, as shown in Figure 2.5.

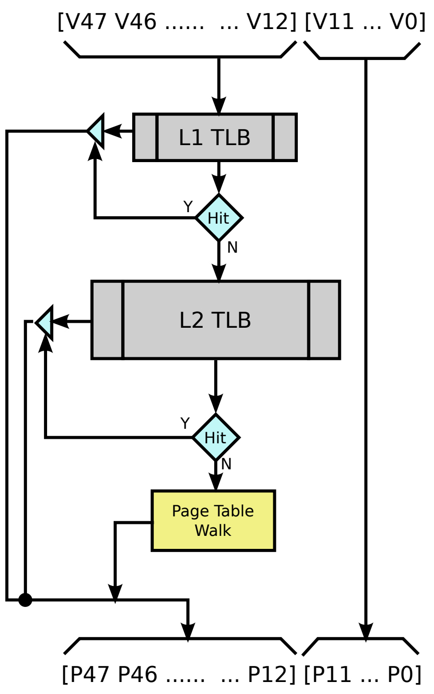

[V47 V46 .... V12] [V11 ... V0]
L1 TLB
Y
Hit
N
L2 TLB
Y
Hit
N
Page Table Walk
[P47 P46 .... P12] [P11 ... P0]

Figure 2.6: To increase TLB reach, processors have employed an hierarchical two-level TLB organization. The L1 TLB is small, set or fully associative, and features a very fast search operation, while the L2 TLB is larger and aims at holding more translations.

In case of a TLB miss, a page walk occurs that fetches from the page table the missing address translation. The page walk can either be performed in hardware (*hardware-managed TLB*) or in software (*software-managed TLB*). The x86-64 architecture uses hardware-managed TLB, and thus the page walk is performed by a hardware finite state machine or simply hardware walker. The hardware state machine walks the page table, finds the missing page table entry, and copies it in the TLB so that future memory references that access the same page will hit in the TLB. During the page walk, the pipeline continues with executing instructions that are independent of the instruction that caused that TLB miss [66, 67].

TLB Reach. The performance of the TLB depends on the *TLB reach*, i.e., the total amount of memory for which the TLB may service address translation requests without experiencing a miss. Because TLB address translation is on the processor's critical path, it requires low access times which constrain TLB size, and thus the TLB reach.

Hierarchical TLBs. The TLB used to be a small monolithic structure with high or fully-associative index methods. Due to the criticality of the TLB in the system's performance and energy, processor vendors have employed an hierarchical two-level TLB organization, similar to the cache hierarchy. Figure 2.6 shows an hierarchical TLB organization: the L1 TLB is small, set or fully associative, and features a very fast search operation, while the L2 TLB is larger and aims at holding more translations. To boost the system's performance further, processors provide separate TLBs for data and instructions. Note that in this thesis we use the term "TLB" to refer either to the general structure of the TLB or to the TLB hierarchy as a whole depending on the context, whereas we use the terms "L1 TLB" or "L2 TLB" to refer explicitly to that (L1 or L2) level's TLBs.

Updating TLB entries. The TLB is a read-only hardware structure that never holds dirty data. When the operating system changes the translation information for a page, e.g., due to relocating that page or due to changing that page's protection access rights, the TLB needs to be updated. This happens with the following procedure: the operating system (i) issues the `INVLPG` instruction to invalidate the stale virtual to physical translation, and (ii) updates the corresponding entries in the page table. Hence, the next memory reference in that page will cause a TLB miss, the hardware state machine will walk the page table and will install correctly the updated translation entry in the TLB.

Summary. The TLB is the hardware key component for accelerating virtual memory. The TLB accelerates address translation and is on the critical path of every memory operation. Due to its important functionality, the TLB performance (hit/miss ratio) affects significantly the total performance [27, 28, 68, 77, 90] and energy consumption [2, 3, 49, 71, 72, 73] of the processor.

2.3.3 MMU cache

To minimize the performance overhead of the TLB misses that trigger page walks, processors have employed MMU caches [23, 28]. The MMU cache reduces the cost of page walks by caching intermediate levels of the page table, i.e., entries from the PML4, PDP, and PD levels.

Figure 2.7 shows the design of MMU cache for the Intel x86-86 processors. The MMU cache is organized into three individual structures; the PD-structure is tagged with the 47-

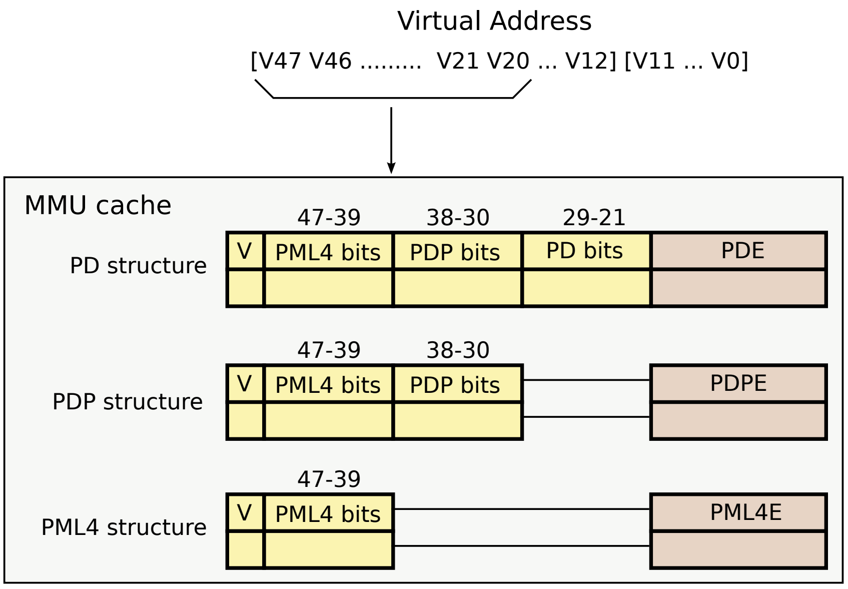


Figure 2.7: The Memory Management Unit (MMU) cache reduces the cost of page walks by caching intermediate levels of the page table.

21 bits, the PDP-structure is tagged with the 47-30 bits, and the PML4-structure is tagged with the 47-39 bits. The hardware walker accesses all structures of the MMU cache with the missing-in-the-TLB virtual address. A hit in the MMU cache enables the hardware walker to skip one or more levels of the page table. The lookup request in the MMU cache may generate three hits, one for each structure; the hardware walker chooses that hit with the longest prefix that enables skipping most levels of the page table. Thus, a page walk requires between one and four memory operations to retrieve the missing address translation, based on the contents of the MMU cache; one memory operation in case of hit in the PD-structure, two memory operations in case of hit in the PDP-structure, three memory operations in case of hit in the PML4-structure, and four memory operations in case of a complete miss in all structures of the MMU cache.

Note that the memory hierarchy may also cache any level of the page table. For example, Intel processors may cache the page table in any level of the memory hierarchy, up to the L1 cache. The difference between the MMU cache and the memory hierarchy in caching page table contents lies in the purpose they serve. The MMU cache defines the number of memory operations per page walk and the hardware walker accesses the memory hierarchy for that many times to retrieve the missing page table entry. The memory

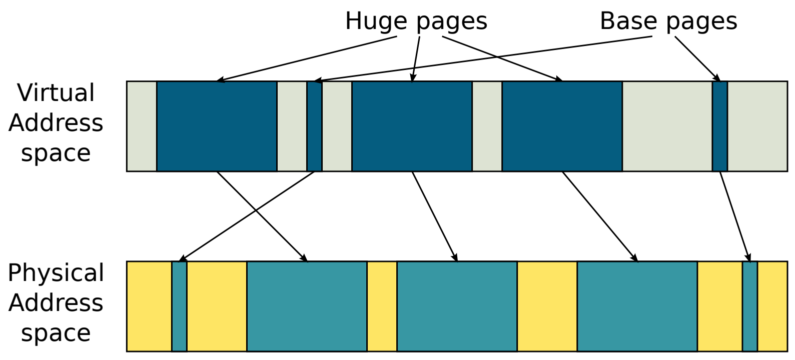


Figure 2.8: Virtual memory abstraction with huge pages.

hierarchy holds close to the processor the contents of the page table to accelerate each page walk reference made by the hardware walker.

2.3.4 Huge Pages

To improve the limited TLB reach that 4 KB pages provide, most architectures provide support for multiple page size through *large* or *huge pages*. For example, the x86-64 architecture supports mixing 4 KB with 2 MB and 1 GB pages, while other architectures support more sizes [91, 101, 107]. Figure 2.8 shows the abstraction of virtual memory that supports both base pages and huge pages.

Huge pages [5, 8] increase the TLB reach by mapping very large regions with a single entry. However, huge pages need to be size-aligned in both virtual and physical address spaces, i.e., a 2 MB mapping may exist only if there is a free 2 MB-aligned page in the virtual address space that can be mapped by a free 2 MB-aligned page in the physical address space.

Page Table Support. Supporting huge pages in an hierarchical page table requires minimal modifications when the available page sizes are defined by the levels of the hierarchical page table. As mentioned before, the x86-64 architecture supports 4 KB, 2 MB, and 1 GB page sizes, and uses a four-level hierarchical tree. The PT entries of the page table hold translation information for 4 KB chunks of memory, the PD entries of the page table hold translation information for 512 * 4 KB = 2 MB chunks, and the PDP entries of the page table hold translation information for 512 * 2 MB = 1 GB chunks. Thus, to store translation information for huge pages, the page table simply holds the address translation in its

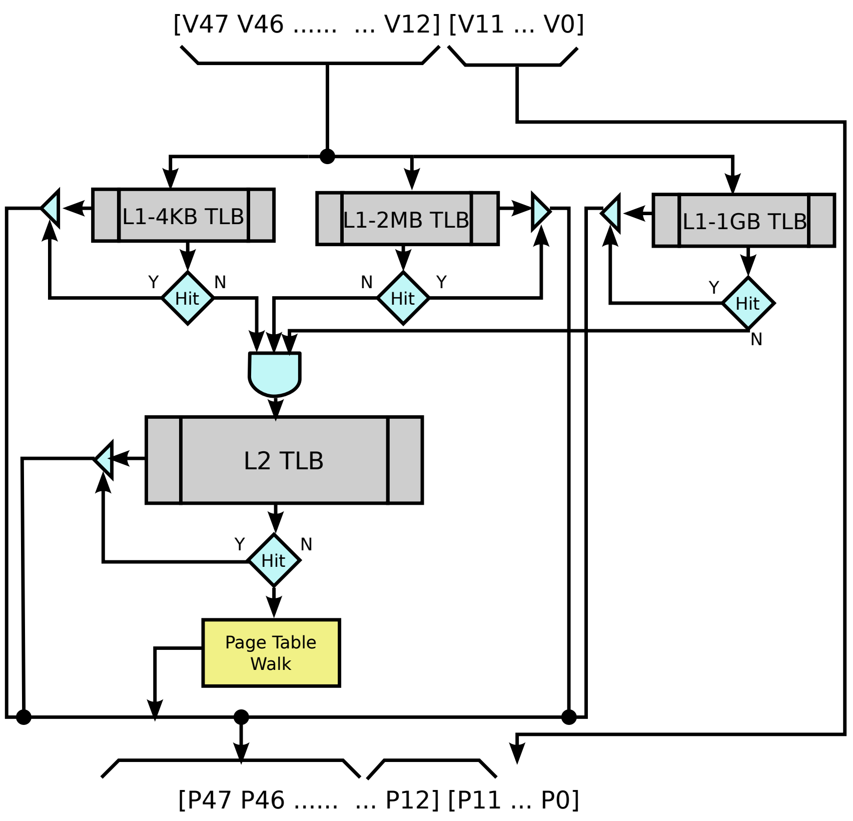

[V47 V46 .... V12] [V11 ... V0]
L1-4KB TLB
L1-2MB TLB
L1-1GB TLB
Hit
Y
N
L2 TLB
Page Table Walk
[P47 P46 .... P12] [P11 ... P0]

Figure 2.9: TLB support for multiple page sizes in Intel x86-64 processors. Separate pages sizes are supported with separate L1 TLBs, with each L1 TLB caching entries for a specific page size.

corresponding level (depending on the page size), instead of having a pointer to the next of the page table.

Note that huge pages may not only increase the TLB reach, but may also reduce the latency of the page walk because less levels in the page table are accessed. A page walk for 4 KB, 2 MB, and 1 GB pages requires up to 4, 3, and 2 memory references, depending on the contents of the MMU cache.

TLB Support. The TLB support for huge pages usually includes either a separate set associative L1 TLB for each page size, as in Intel processors [56] and shown in Figure 2.8, or a single fully associative L1 TLB that supports both 4 KB and huge pages, as in SPARC and AMD processors [14, 107]. These two approaches dominate because supporting all page sizes in a single set associative TLB is not straight-forward: the page size defines the index bits to access the TLB, but the page size is unknown during the TLB lookup time [95, 112].

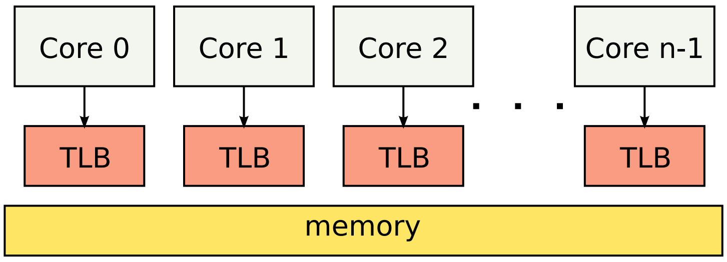


Figure 2.10: In a multicore system, each core is equipped with a private TLB hierarchy.

2.4 Address Translation in the Multicore Era

In a multicore system, each core is equipped with a *private TLB hierarchy* (Figure 2.10). In contrast to the memory hierarchy of a cache-coherent system, the TLBs are read-only structures and, thus, lack hardware support for coherency.

TLB Shootdown. Similarly to when the translation information for a page changes in a single-core system, it is again the operating system that is responsible for keeping the TLBs coherent and consistent in a multicore system. This happens with a process known as *TLB shootdown* [34].

The TLB shootdown is a two-phase commit transaction that ensures that all cores evict the affected mapping that may currently hold in their TLBs. The operating system first locks the affected page table entry, creates a list of cores that could hold the affected mapping, and sends expensive inter-processor interrupts to notify those cores to invalidate that mapping. Because the operating system lacks precise information about which mappings each TLB holds, it usually notifies all cores—interrupting some of them falsely—to ensure correctness. After all involved cores acknowledge the invalidation in their private TLB hierarchy, the operating system continues with updating the corresponding page table entry and releasing the lock. A future memory reference that accesses that page will trigger a TLB miss, and the page walker will fetch the updated entry from the page table.

2.5 Accessing memory with virtual memory

The TLB is on the critical path of the processor for accessing the cache hierarchy. Figure 2.11 shows the three different design points in the co-organization of the TLB and

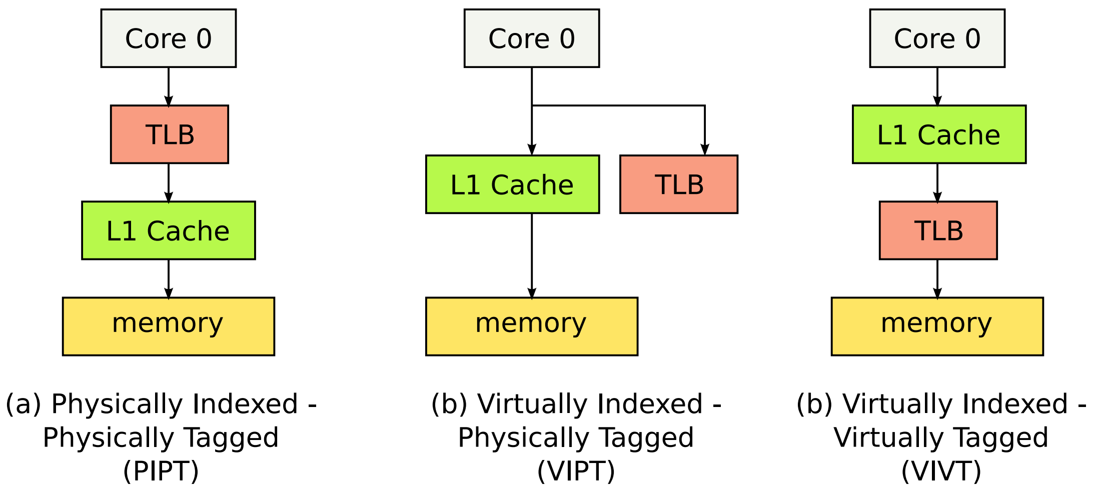

Core 0
TLB
L1 Cache
memory
(a) Physically Indexed - Physically Tagged (PIPT)
Core 0
L1 Cache
TLB
memory
(b) Virtually Indexed - Physically Tagged (VIPT)
Core 0
L1 Cache
TLB
memory
(b) Virtually Indexed - Virtually Tagged (VIVT)

Figure 2.11: There are three possible configurations for the processor to access the memory hierarchy with virtual memory: (a) Physically-indexed/physically-tagged (VIPT) caches, (b) Virtually-indexed/Physically-tagged (VIPT) caches, and (c) Virtually-indexed/Virtually-tagged (VIVT) caches or Virtual caches.

cache hierarchy: (i) the physically-indexed/physically-tagged caches, (ii) the virtually-indexed/physically-tagged caches, and (iii) the virtually-indexed/virtually-tagged (VIVT) caches or virtual caches. Next we explain in more detail the role of the TLB in these designs.

Physically-indexed/physically-tagged (PIPT) caches use the physical address for both the index and the tag bits [58]. The processor accesses first the TLB to obtain the physical address and then the cache. PIPT caches form a simple cache organization and avoid problems with *synonyms*, i.e., when multiple virtual pages are mapped to the same physical page. However, the latency of the cache access depends directly on the latency of the TLB lookup, even in the case of TLB hit, limiting thus TLB sizes.

Virtually-indexed/Physically-tagged (VIPT) caches use part of the virtual address for the index bits and the physical address for the tag bits [58]. The processor accesses the cache and the TLB in parallel. Thus, a VIPT cache can achieve lower latency compared to a PIPT cache, as the cache line can be looked up in parallel with the TLB translation. VIPT caches form a faster cache organization than PIPT caches and avoid problems with *homonyms*, i.e., when the same virtual page is mapped to multiple physical pages. However, similar to the PIPT cache organization, the TLB lookup still needs to be fast because the cache tag cannot be compared until the physical address is available, and thus the TLB size remains limited. Note that the PIPT and VIPT organizations are the most common in commodity processors.

Virtually-indexed/Virtually-tagged (VIVT) caches or Virtual caches form an alternative option that removes the address translation from the critical path in accessing memory [26, 36, 37, 54, 65, 80, 100, 116, 120]. Virtual caches use virtual addresses to access the cache hierarchy down to a certain level and only consult the TLB on a cache miss beyond the supported level in the cache hierarchy. Although virtual caches reduce the performance and energy overheads of the TLB by only translating after a cache miss, ensuring correct execution requires extra hardware support and complexity for handling synonyms, homonyms, coherence, and protection access rights. In addition, for workloads that suffer many TLB misses due to poor locality, virtual caches just shift the translation to a lower level of the cache hierarchy.

2.6 Segmented Virtual Memory

In the previous section we described the basic concepts and architectural support for page-based virtual memory. In this section we discuss segmentation that is another approach for providing and implementing virtual memory.

Segmentation. With segmented virtual memory, the virtual and physical address spaces are divided and managed into segments of arbitrary length, instead of fixed-size pages as with paged virtual memory.

Segments typically correspond to parts of a process such as the stack, the heap, and code sections. A process may create multiple segments for multiple program modules, or for multiple classes of memory usage such as code and data segments. Each segment is identified with a segment identifier, and holds information related to the segment, such as its length (the *limit*) and permissions access rights. In the case of pure segmentation, each segment holds also address translation information (the *offset*), as explained next.

The primary roles of segmentation is to enforce memory protection and to enhance memory sharing between multiple processes or multiple modules of a single process. With segmentation, a process is allowed to make a reference into a segment, only if the type of reference is allowed by the permissions, and if the offset lies within the segment.

However, one of the important differences between segmented and paged virtual memory is that segmentation is visible to the processes, as part of the memory model semantics. Hence, segmentation structures memory into multiple spaces, removing the “illusion” of a single large space that paged virtual memory provides.

Pure Segmentation. Some commercial processors have used pure segmentation without paging to implement virtual memory, such as the Burroughs B5000 [85], the 8086 [4], and iAPX 432 [60] processors. Each segment holds the *base*, i.e., information that indicates where the segment is located in memory. When a program references a memory location, the offset is added to the segment base to generate a physical memory address.

Although pure segmentation seems a good approach for reducing the overheads of virtual memory, it suffers from various drawbacks. Pure segmentation requires that entire segments be swapped back and forth between the physical memory and the backing storage. Furthermore, the memory management becomes complex for the operating system. When a process requests to allocate a segment, the operating system has to allocate enough contiguous free memory to hold the entire segment. This often results in external memory fragmentation.

Paged Segmentation. To overcome these limitations, some architectures provide paged segmentation or segmentation over paging, such the PowerPC and the x86 architectures [66, 67]. In paged segmentation, segments serve only memory protection and sharing purposes. However, segments are backed by pages as well. Thus, address translation still occurs in page granularity.

With paged segmentation, the operating system only moves individual pages between main memory and backing storage, similar to page-based virtual memory. Pages of the segment can be located anywhere in main memory and need not be contiguous, and thus the memory fragmentation is reduced.

In the x86 architecture, the segmentation adds one more level in the address translation path. The application issues memory references using the virtual address space. A virtual address is translated first to a *linear address* using the segmentation support, i.e., a few (e.g., 6) segment registers, and then the linear address is translated to physical address using the paging support, i.e., the TLB and the page table, as in paged virtual memory.

The x86-64 architecture does not use segmentation over paging. Four of the six segment registers (CS, SS, DS, and ES) are hard-coded and ignored, and only two segment registers (FS and GS) are used by the operating system for special purposes.

3
Quantifying Address Translation Performance Overheads

3.1 Introduction

In recent years, companies like Amazon, Google and Facebook have invested resources to build big datacenters where their software infrastructure runs on a large number of inexpensive computers. The datacenters aim to provide the most scalable and economical way to leverage the vast amount of available processing power. Given the high cost of building and maintaining datacenters, a single-digit performance improvement in the utilization of datacenters translates directly into savings in money. To this end, datacenter infrastructures have received attention during the last years in improving the performance of all the involved components such as processors, storage, and interconnection networks.

To stimulate the research in the topic of datacenters, the CloudSuite benchmark suite was recently introduced [50]. CloudSuite is a collection of popular scale-out applications that target various domains of datacenter computing including data analytics (MapReduce), key-value caching (MemCached) and storing (NoSQL), large-scale graph analytics (GraphLab), and web-searching (Nutch) among others. Scale-out applications operate on large datasets with low memory locality exhibiting inefficient execution in traditional server architectures [50]. In response, computer architects proposed novel designs to increase the efficiency of microprocessors for scale-out applications through improvements in the processor pipeline [87], the memory hierarchy [69], and the on-chip interconnection network [86].

However, the overhead of address translation in the Memory Management Unit (MMU) for scale-out applications has been largely ignored. There have been very few studies on the performance cost of the MMU that proposed solutions to mitigate them through either reducing the number of TLB misses [27] or the cost of page walks [28]. Still, these studies did not provide an extensive characterization of the MMU behavior in the context of datacenter computing.

Our goal in this chapter is to understand how the MMU (i) performs under the execution of scale-out applications, (ii) affects the application performance, (iii) interacts with other components of the processor, and (iv) can be potentially improved to boost the performance of datacenters. To this end we analyze the performance of the Memory Management Unit under the execution of various scale-out applications. We conduct our analysis leveraging the use of performance counters on an x86-64 real system.

In summary, the main contributions of this chapter are:

- We perform a comprehensive performance analysis of the MMU for several scale-out applications showing that the MMU overhead accounts up to 16% of the total execution time.

- We find that by reducing the MMU overheads, the performance improves by up to 13.9% enabling better exploitation of the available execution resources.

- We observe that huge pages are beneficial for most applications without being an "always-win" option due to limited hardware support.

- We quantify the interference between the application data and the page-table structures in the cache hierarchy, and show how page walks are affected by hardware prefetchers.

- We present upper-bound analyses and provide potential directions for improving the MMU performance.

In Section 3.2 we provide background information regarding the scale-out applications that we use in our study, while in Section 3.3 we explain our methodology. We present the performance analysis of the MMU under the execution of the scale-out workloads in Section 3.4, and we discuss potentials for improving the MMU performance in Section 3.5. Finally, in Section 3.6 we review the related work and in Section 3.7 we conclude our study.

3.2 Background

In this section we briefly describe the scale-out applications from CloudSuite [6, 50] that we use in our study. Note that Chapter 2 provides background information about the hardware support of the MMU—the Translation Lookaside Buffer (TLB) and the MMU cache—of the x86-64 architecture which constitutes the dominant processor architecture deployed in today's datacenters [25].

3.2.1 Scale-out Applications

Scale-out computing increases the computational power of a datacenter in an horizontal fashion by adding more inexpensive nodes to the datacenter, in contrast to scale-up computing [18]. Typically, these nodes are based on commodity components and connected through high-performance inter-connection networks. Such datacenter infrastructures target the execution of scale-out applications that: (i) operate on large data sets that are split across nodes, (ii) serve independent requests exhibiting very low inter-node sharing, and (iii) are designed for datacenters with unreliable nodes. Next we briefly describe the scale-out applications that we use in our study.

Data-analytics (MapReduce). This benchmark uses Mahout, a scalable machine learning and data mining library designed for the Hadoop MapReduce framework. The benchmark performs the Bayesian classification algorithm for a large input set of Wikipedia articles.

Data-caching (MemCached). MemCached is a distributed memory caching system that speeds up dynamic database-driven websites by caching data in main memory to reduce the number of accesses in the database. The benchmark simulates the behavior of a caching server for Twitter.

Data-serving (NoSQL). This benchmark targets the domain of NoSQL databases which have gained growing industry use in big data and real-time web applications. The benchmark uses Cassandra, a column-oriented database server, and simulates an update-heavy workload.

Graph-analytics (GraphLab). This benchmark relies on GraphLab, an abstraction framework that expresses asynchronous, dynamic, graph-parallel computation. The benchmark is a GraphLab-based implementation of *tunkrank* that measures a person's influence on Twitter.

Media-streaming (QuickTime). This benchmark targets the domain of media-streaming services and uses the DarwinStreaming Server (open-source equivalent of Apple QuickTime Server) that streams media to clients across the Internet.

Software-testing (Cloud9). This benchmark uses Cloud9, an automated software-testing platform that parallelizes symbolic execution.

Web-search (Nutch). This benchmark targets web search engines that dominate among the internet services [70]. The benchmark uses the distributed version of Nutch, an open source web search engine, with content crawled from *http://en.wikipedia.org/*.

3.3 Methodology

Here we describe the experimental environment and the methodology we followed to analyze the MMU performance.

3.3.1 System Setup

We conduct our study on a 4-core Intel Xeon E3-1230 (Sandy Bridge) running at 3.2 GHz with hyper-threading enabled and equipped with 16 GB memory. Each core has a private TLB hierarchy, as shown in Table 3.1: an L1 dTLB for data accesses, an L1 iTLB for instruction accesses, and a unified L2 TLB, i.e. shared between the L1 iTLB and L1 dTLB [62].

Per-core TLB Hierarchy			
L1 iTLB	4 KB	128 entries	4-way assoc.
	2 MB	8 entries	fully assoc.
L1 dTLB	4 KB	64 entries	4-way assoc.
	2 MB	32 entries	4-way assoc.
L2 TLB	4 KB	512-entries	4-way assoc.
	2 MB	—	

Table 3.1: TLB hierarchy of the test machine.

Note that in this chapter we focus on the impact of L2 TLB misses and L1-2MB TLB misses, i.e., misses to 2 MB pages, both of which trigger page walks.

The system runs OpenSuse 12.3 with the 3.7.10-1.4 Linux kernel. We used seven out of the eight scale-out applications from CloudSuite [6]; we faced tuning problems with *web-serving*. For all the server-oriented applications (*data-caching*, *data-serving*, *web-serving*, and *web-search*), we set up both clients and servers on the same machine pinning each to unique cores through the *taskset* utility and we measured only the activity of the server programs. Finally, to access the performance counters we use the *perf* utility [11] and we report the average results of three runs. Table 3.2 summarizes the performance events and the metrics that we use. Because our metrics require getting information about more performance events than the available hardware performance counters that our machine provides, therefore we run multiple times for a separate set of performance events to avoid multiplexing and to get accurate measurements. Note that our machine lacks TLB support for 1 GB pages and support for counting performance events related to the MMU cache. Consequently we limit our evaluation of varying the page-size to 4 KB and 2 MB, and do not present performance events for the MMU cache.

3.3.2 Huge Pages

Linux provides two mechanisms for enabling huge 2 MB pages: (i) Transparent Huge Pages (THP) [5] and (ii) libhugetlbfs [8]. THP attempts to allocate huge pages to service application's memory requests that are naturally 2 MB-aligned in the anticipation of subsequent memory allocations. If no huge pages are available, the kernel falls back to 4 KB pages, and periodically scans through the memory to substitute several 4 KB pages with a huge page. On the other hand, with libhugetlbfs [8], huge pages must be set aside at boot

Equations & Performance events		
(%) Cycles spent in page walks due to data accesses	=	(DTLB_LOAD_MISSES.WALK_DURATION + DTLB_STORE_MISSES.WALK_DURATION) / CPU_CLK_UNHALTED.THREAD_P
Page walks per 1000 instr. due to data accesses	=	(DTLB_LOAD_MISSES.WALK_COMPLETED + DTLB_STORE_MISSES.WALK_COMPLETED) / (INST_RETIRED.ANY_P / 1000)
Average cycles per page walk due to data accesses	=	(DTLB_LOAD_MISSES.WALK_DURATION + DTLB_STORE_MISSES.WALK_DURATION) / (DTLB_LOAD_MISSES.WALK_COMPLETED + DTLB_STORE_MISSES.WALK_COMPLETED)
(%) Cycles spent in page walks due to instruction accesses	=	(ITLB_MISSES.WALK_DURATION / CPU_CLK_UNHALTED.THREAD_P)
Page walks per 1000 instr. due to instruction accesses	=	(ITLB_MISSES.WALK_COMPLETED * 1000) / INST_RETIRED.ANY_P
Average cycles per page walk due to instruction accesses	=	(ITLB_MISSES.WALK_DURATION / ITLB_MISSES.WALK_COMPLETED)
L1 Cache misses	=	L1D.REPLACEMENT
L2 Cache misses	=	(MEM_LOAD_UOPS_RETIRED.LLC_HIT + MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT + MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM + MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS)
Last-level Cache (LLC) misses	=	MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS

Table 3.2: Metrics and corresponding performance events based on hardware performance counters that we use in Section 3.4. The naming of performance events is according to [63].

Benchmark	Working Set	Anonymous Total Pages %	Anonymous Huge Pages %
Data-analytics	5 GB	99.84	72.53
Data-caching	8 GB	99.91	99.89
Data-serving	7 GB	46.42	37.61
Graph-analytics	12 GB	99.89	62.69
Media-streaming	700 MB	99.82	—
Software-testing	700 MB	97.11	25.67
Web-search	6 GB	99.37	75.07

Table 3.3: Memory usage statistics. The first column shows the size of the working set, the second column indicates the percentage of allocated anonymous pages over the working set, and the third column shows the percentage of anonymous pages that were allocated as huge pages with THP.

time and they are not swappable. There are two important differences between THP and libhugetlbfs. The first difference is that THP supports 2 MB pages transparently to application, whereas libhugetlbfs requires that the application must explicitly request 2 MB pages during memory allocation. The second difference between the two mechanisms is that THP supports 2 MB pages only for anonymous pages, i.e., pages that are not backed by or associated with files, while libhugetlbfs supports 2 MB pages for memory-mapped files as well.

We use the following methodology to decide which mechanism we should enable in this study. We run the scale-out applications only with THP enabled and we periodically collect memory statistics from the *proc* filesystem [9] regarding (i) the active working set, (ii) the percentage of the allocated pages that are anonymous, and (iii) the percentage of the allocated huge pages over the total working set. Table 3.3 summarizes the results.

We observe that most scale-out applications use anonymous pages for more than 96% of their working set. The exception is *data-serving* whose working set is mainly divided among the java heap that uses anonymous pages (46%) and the NoSQL database that is memory-mapped. Regarding the ability of THP to successfully allocate huge pages, we find that huge pages cover: (i) 72% for *data-analytics*, 99% for *data-caching*, and 75% for *web-search*, (ii) more than 62% for *graph-analytics*, (iii) only the java heap (37%) for *data-serving*, (iv) 25% for *software-testing* and surprisingly 0% for *media-streaming*. These results indicate that THP are able to provide huge pages for most of the scale-out applications. Finally, to get more confidence about our execution environment we run the applications with libhugetlbfs. We find that the performance is similar among the two configurations for all

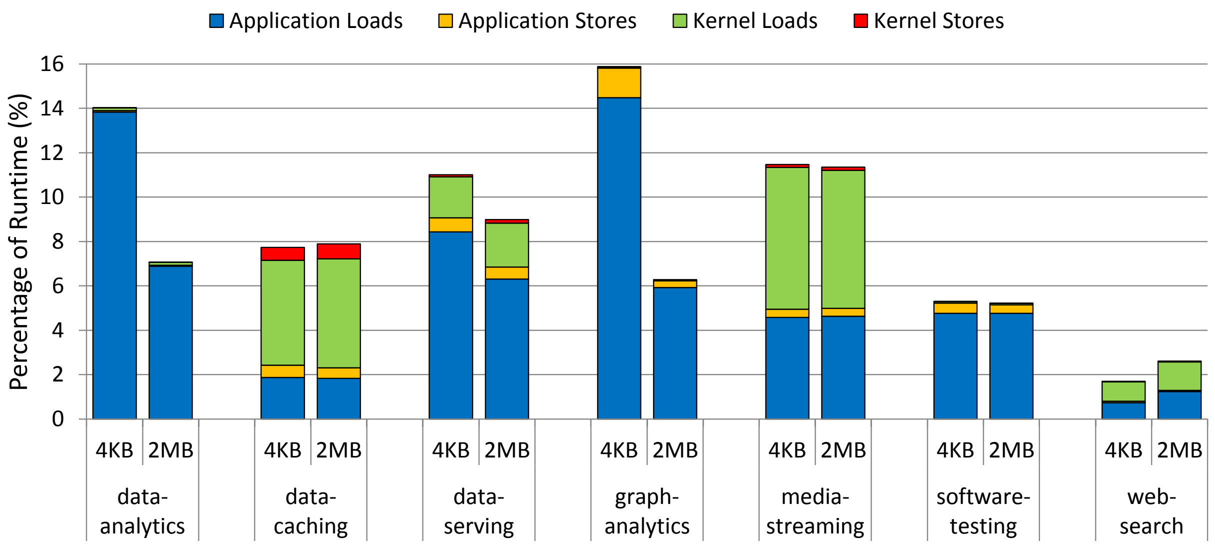


Figure 3.1: Percentage of execution time spent in page walks due to data accesses with 4 KB and 2 MB pages. The MMU overhead accounts up to 16% of the total execution time.

applications, including *data-serving* and *media-streaming*, after spending significant effort in tuning libhugetlbfs for the needs of each application. Thus, we decide to use 2 MB pages through Transparent Huge Pages.

3.4 MMU Performance Analysis

In this chapter we analyze the performance of the Memory Management Unit (MMU) under the execution of scale-out applications. We mainly focus on the data accesses that typically stress the MMU more than the instruction accesses [29, 75]. We measure the overhead due to page walks and its impact on the application's performance, we quantify how often a page walk occurs in terms of TLB misses per 1000 instructions (MPKI), and we report the average cost of a page walk. Moreover, we evaluate the interference between the application data and the page walks in the cache hierarchy, we show how the cache hardware prefetchers affect the MMU performance, and we discuss the performance of the MMU for instruction accesses. Finally, we summarize the key findings and their implications in the MMU performance.

3.4.1 How much time is spent in TLB misses?

The MMU overhead is dictated by the time spent in TLB misses that trigger page walks, because short L1 TLB misses that hit in the L2 TLB may be overlapped with execution. Figure 3.1 shows the percentage of the execution time spent in page walks due to data accesses with 4 KB pages (left bar). We make the following observations.

First, we find that all applications suffer from high MMU overheads with 4 KB pages. More specifically, *data-serving* and *media-serving* spend more than 10% of the execution time in page walks, while *data-analytics* and *graph-analytics* reach almost 14% and 16%, respectively. These applications operate on large datasets (Table 3.3) with low locality [50] stressing the performance of the MMU. To confirm this behavior, we also calculate the number of cold TLB misses based on the working set and the page-size. We find that the cold TLB misses contribute less than 0.02% to the total TLB misses for all the scale-out applications. These results indicate that the MMU overhead is practically dictated by capacity and conflict TLB misses due to the limited MMU resources and the low memory locality of the workloads, rather than by cold TLB misses.

Second, we find that the page walks due to kernel code contribute significantly to the total MMU overhead for *data-caching*, *data-serving*, *media-streaming* and *web-search*. The reason is that these applications stress the network and the file-system stack [50, 84]. Indeed we find that *data-caching*, *data-serving*, *media-streaming*, and *web-search* spend 68.6%, 25.5%, 67.2%, and 10.7% of the total execution time in kernel code, respectively.

We analyze the kernel page walks and categorize them according to the execution code path. More than 85% of the kernel page walks take place in functions due to both file-system and network activity for *data-caching*, *media-streaming*, and *web-search*, while 44.3% accounts to both file-system and scheduler/synchronization activities for *data-serving*. Moreover, we identify two hotspot functions responsible for page walks: 7.1% for *data-caching* and 16.9% for *media-streaming* of total page walks occur only in the kernel function `tcp_poll()` due to network activity, and 5.7% for *data-caching*, 14.3% for *media-streaming* and 9.1% for *web-search* only in `fget_light()` due to both network and file-system activity.

Findings

- The MMU overhead for the scale-out applications is significantly high up to 16%.

- The kernel page walks may contribute more than 50% of the total MMU overhead mainly due to network and file-system activity.

3.4.2 Do Huge Pages help?

To observe the impact of the page-size in the performance of the MMU, we leverage Transparent Huge Pages (THP) that enable 2 MB pages when possible and fall back to 4 KB pages, as explained in Section 3.3.2. Depending on the application behavior, huge pages may decrease the time spent in page walks due to: (i) reducing the number of TLB misses by increasing the TLB reach (Section 3.4.4), and/or (ii) reducing the average cost of page walk by requiring one memory reference less (Section 3.4.5). Figure 3.2 shows the cycles spent in page walks due to the employment of 2 MB pages normalized to the page walk cycles with 4 KB pages.

First, we notice that increasing the page-size reduces mostly the MMU overhead by up to 65.9% for *data-analytics*, *data-serving* and *graph-analytics*. *Data-serving* gets the least benefits from 2 MB pages among these three applications (23.6%). The reason is that *data-serving* accesses the NoSQL database through memory-mapped files. However, THP lack support for memory-mapped files, as discussed in Section 3.3.1. Consequently, the THP mechanism covers with 2 MB pages less part of the working set (only the java heap) for *data-serving* than for *data-analytics* and *graph-analytics* (Table 3.3).

Second, we notice that huge pages reduce slightly the time spent in page walks by less than 2% for *data-caching*. We find that the number of page walk cycles due to user code with huge pages decreases by 31% on one hand. On the other hand, the number of the dominant page walks cycles due to kernel code increases by 19%. The reason for this behavior is the combination of the application's poor locality with the increased pressure on the TLB for 2 MB pages: (i) the TLB supports only 32 entries for 2 MB pages, (ii) the kernel typically uses 2 MB pages for its internal structures [5, 46], (iii) with THP enabled there is contention between user-level and kernel-level TLB entries in the limited-sized TLB for 2 MB pages, since x86-64 architecture does not lock TLB entries for kernel usage. Consequently, the time spent in page walks for the application data decreases at the cost of increasing the kernel overheads.

Third, we notice that for *web-search*, increasing the page-size actually increases the time spent in page walks by 54%. Similarly to *data-caching*, the reason for this behavior

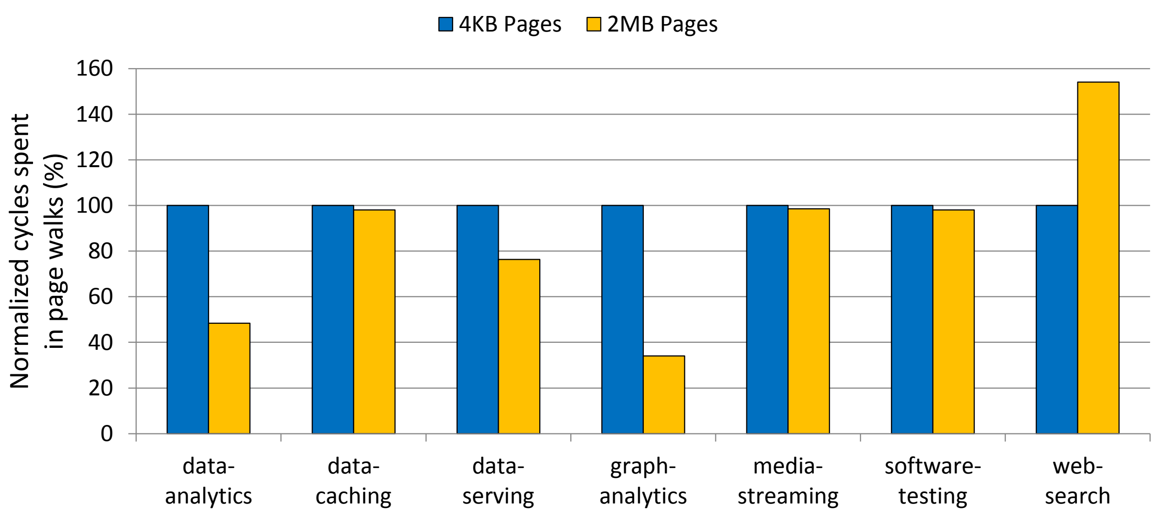


Figure 3.2: Normalized cycles spent in page walks due to data accesses with 4 KB pages and 2 MB pages.

is the limited TLB support for 2 MB pages in combination with the low data locality of the application.

Fourth, 2 MB pages bring negligible benefits for *media-streaming* and *software-testing*. Surprisingly we found THP fail to allocate any 2 MB pages for *media-streaming*. The reason is that this scale-out application performs a small number of memory allocations (`mmap()`) with such arguments, i.e., size, flags, and access rights, that do not allow the THP mechanism to allocate huge pages. Finally, the reduction of page walk cycles for *software-testing* is limited by the ability of THP to back only 25.7% of the application's working set with 2 MB pages.

Figure 3.1 shows the percentage of execution time spent in TLB misses that trigger page walks with 2 MB pages (right bar). This percentage is now computed based on the total execution time with 2 MB pages (we discuss the actual performance differences in Section 3.4.3). We observe that an important fraction of time—more than 6%—is still spent in page walks even for those applications that benefit from 2 MB pages (e.g. *data-analytics* and *graph-analytics*). For the rest of the applications the percentage remains practically the same. These results indicate that in case the application performance depends directly on any improvements in the MMU performance, there is still ample space for optimizing the MMU.

Findings

- Huge pages reduce the MMU overhead for some scale-out applications by up to 65.9%. However, the limited hardware support for huge pages may actually increase the MMU overhead by up to 54%.
- Huge pages put more pressure on the TLB for those applications that suffer from a high number of kernel page walks due to the limited support for 2 MB.
- The software implementation of some scale-out applications cannot use huge pages.
- Even if huge pages benefit the application performance, still a significant percentage of time is spent in page walks leaving space for optimizations.

3.4.3 Do TLB misses affect performance?

In this section we quantify the application speedup due to improving the MMU performance by changing the page-size from 4 KB to 2 MB. To assess the importance of the MMU performance in the processor pipeline, we compute also the *expected performance* with 2 MB pages based on Equation 3.1 [27]. The expected performance with 2 MB pages is computed as the measured number of execution cycles with 4 KB pages reduced by the measured improvement in cycles spent in page walks when increasing the page-size from 4 KB to 2 MB. In other words, the expected performance assumes that the page walks do not affect at all (neither positively nor negatively) the processor pipeline (out-of-order execution, memory hierarchy, etc.).

$$ExpectedTotalCycles_{2M} = TotalCycles_{4K} - TlbCycles_{4K} + TlbCycles_{2M} \tag{3.1}$$

Figure 3.3 shows the measured speedup that is achieved due to employing huge pages on the left bar, and the expected speedup based on Equation 3.1 on the right bar.

Regarding the measured speedup we see that the performance increases for those applications that reduce the time spent in page walks (Figure 3.2). More specifically, 2 MB pages boost the performance of *data-analytics*, *data-serving*, and *graph-analytics* by 9.7%, 6.4% and 13.9% respectively. The rest of the applications achieve negligible speedup, while the

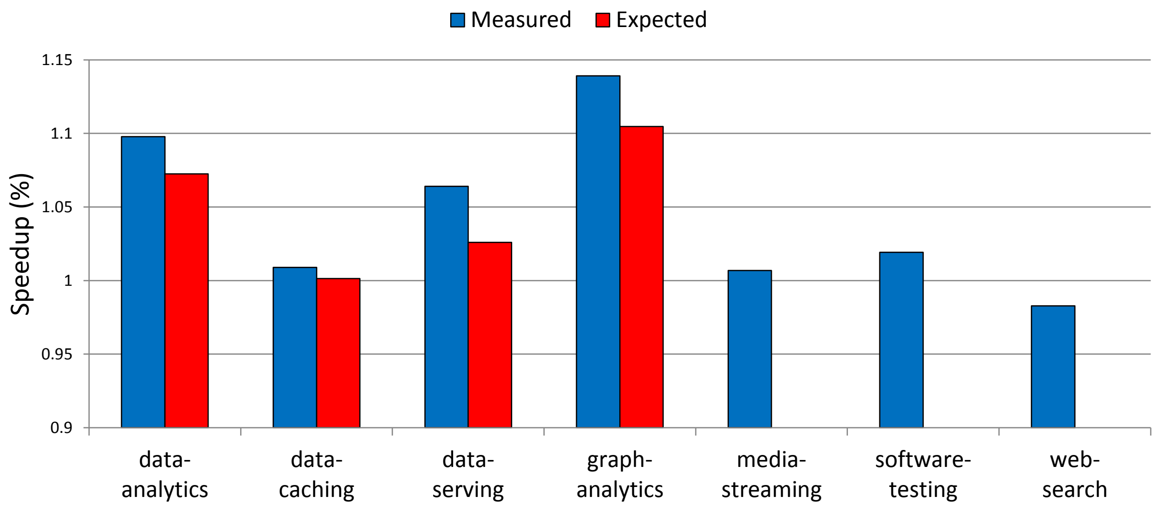


Figure 3.3: Speedup due to increasing the page-size from 4 KB to 2 MB. The left bar corresponds to the actual measurements, while the right bar corresponds to the expected speedup based on Equation 3.1. The difference indicates the positive impact of reducing the MMU overhead in the processor pipeline. Note that we do not report expected speedup for those applications that use throughput as performance metric, i.e. *media-streaming*, *software-testing* and *web-search*.

performance of *web-search* slightly drops by 2% as expected due to spending more cycles in page walks.

Regarding the expected speedup we notice a positive gap between the expected speedup and the measured one, i.e., 9.7% vs. 7.2% for *data-analytics*, 6.4% vs. 2.6% for *data-serving* and 13.9% vs. 10.5% for *graph-analytics*. We believe that this difference is due to the impact of page walks on the out-of-order and hyper-threading execution, and the memory hierarchy. The page walks occur less frequently and need less cycles to complete with huge pages, allowing better utilization of the available pipeline resources. To verify this behavior, we measure also the number of stalled cycles in the back-end of the processor pipeline. We find that by using 2 MB pages, the number of back-end stalled cycles reduces by 16.7% for *data-analytics*, by 10.7% for *data-serving*, and by 10.1% for *graph-analytics*, and we also find that the IPC increases by 12.3% for *data-analytics*, by 8.1% for *data-serving*, and by 7.4% for *graph-analytics*. We also observe fewer cache misses with 2 MB pages as we will explain next in Section 3.4.7. Our results suggest that future improvements in MMU performance will allow better exploitation of the available execution resources.

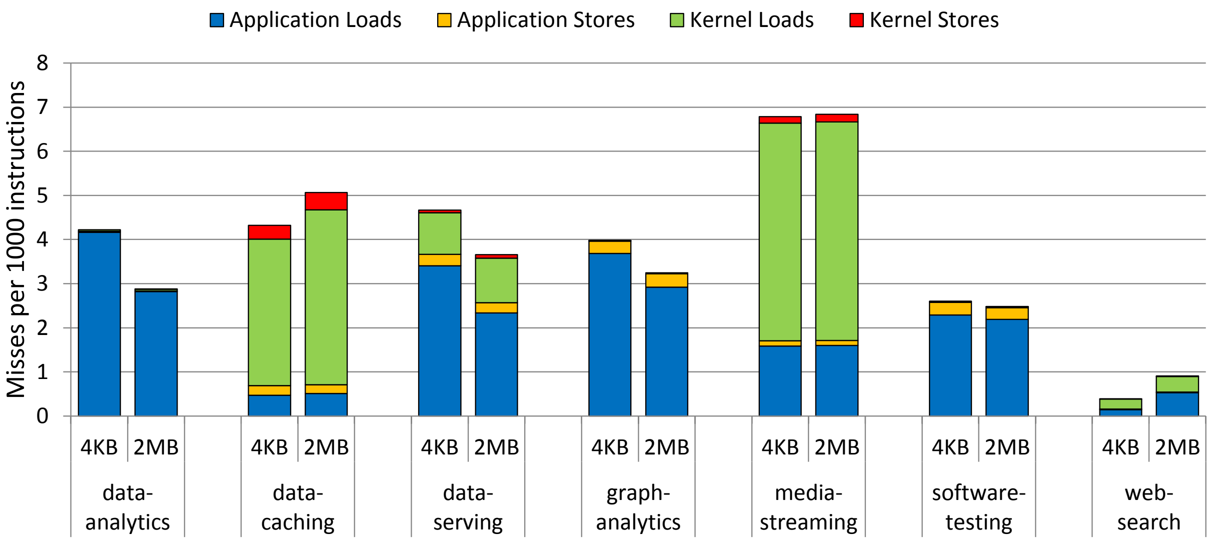


Figure 3.4: Page walks (i.e. TLB misses) per 1000 instructions (MPKI) due to data accesses with 4 KB and 2 MB pages.

Findings

- Improved MMU performance speeds up the scale-out applications by up to 13.9%. However, the limited hardware support for 2 MB pages may reduce the application performance by 2%.
- The difference between the expected and the measured application speedup indicates that optimizations in the MMU will result in more efficient utilization of the execution pipeline.

3.4.4 How often do TLB misses occur?

Figure 3.4 shows the number of TLB misses that trigger page walks per thousand instructions (MPKI) due to data accesses when using 4 KB and 2 MB pages. By changing the page-size from 4 KB to 2 MB, the MPKI reduces for those applications that the page walk overhead decreases (e.g. *data-analytics*, *data-serving* and *graph-analytics*), but not in the same ratio as in Figure 3.1 since 2 MB pages reduce also the cost per TLB miss as we explain in the following subsection. However, we notice that the MPKI actually increases for *data-caching* and *web-search* with 2 MB pages. This behavior confirms the limited hardware support for 2 MB in current MMUs.

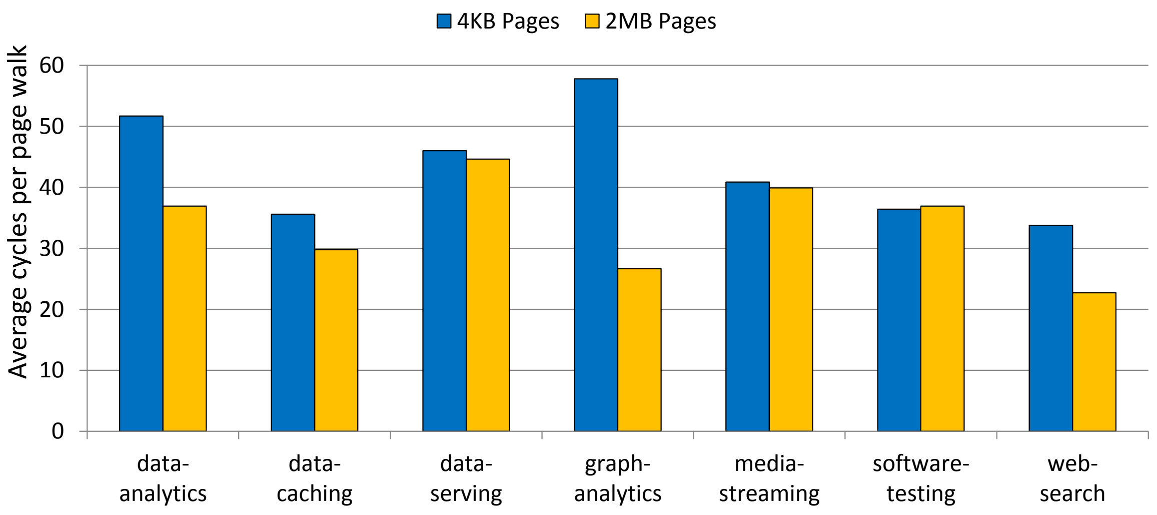


Figure 3.5: Average number of cycles per page walk due to data accesses with 4 KB and 2 MB pages.

Findings

- The frequency of page walks decreases for some applications due to huge pages. However the limited TLB support for 2 MB pages may increase the MPKI.

3.4.5 What is the cost of a TLB miss?

Figure 3.5 shows the average cost of a TLB miss that triggers a page walk with 4 KB and 2 MB pages. We observe that the average cost of page walk with 4 KB pages is far lower than 100 cycles for all applications. This indicates that resolving a page walk does not require any off-chip memory access on average. Our results corroborate previous studies [23, 89] that focused on different applications and concluded that the page walks references typically hit in the cache hierarchy.

The latency of the average cost per page walk depends on (i) the performance of the MMU cache, which dictates how many memory accesses (up to four) are necessary to resolve the page walk, and (ii) the locality of the page table references in the data cache hierarchy (i.e., L1, L2, or LLC). Unfortunately, our experimental machine does not provide any performance events for measuring the behavior of the MMU cache. However, we perform an upper-bound analysis of perfect MMU caches in Section 3.5.2, and we draw some conclusions about the locality of the page table references in the cache hierarchy.

By comparing the results for the two page-size configurations, we observe that the

Benchmark	Page walks per 1000 instr.	Average cycles per page walk	Time spent in page walks
Data-analytics	0.68	0.71	0.48
Data-caching	1.17	0.83	0.98
Data-serving	0.78	0.97	0.76
Graph-analytics	0.81	0.46	0.34
Media-streaming	1.01	0.98	0.99
Software-testing	0.95	1.01	0.98
Web-search	2.32	0.67	1.54

Table 3.4: Summarized results for page walks per 1000 instructions, average cycles per page walk, and cycles spent in page walks with 2 MB pages, normalized to 4 KB pages (lower is better).

average cost per page walk is lower for most scale-out applications with 2 MB pages. In conjunction with the results of the previous sections, which are summarized in Table 3.4, we observe that the average cost of a page walk with 2 MB pages reduces significantly for *data-analytics* and *graph-analytics* as expected. For *data-caching* and *web-search*, the average cost also decreases and compensates the increase in MPKI, while for *data-serving*, *media-streaming*, and *software-testing*, that have low use of 2 MB pages, the average cost remains practically the same.

Findings

- The average TLB miss cost indicates that page walk references typically hit in the data cache hierarchy.

3.4.6 Comparison with other benchmark suites

In this section we compare the performance of the MMU across different benchmark suites. Figure 3.6 shows the percentage of execution time spent in page walks for SPEC 2006 [12], BioBench [15], Parsec [32], and CloudSuite.

We observe that the scale-out applications consistently stress the MMU more compared to other benchmark suites in terms of runtime overhead. Moreover, we find that scale-out applications suffer almost an order of magnitude more frequently from page walks than other benchmarks; the number of page walks per 1000 instructions is well below 1 for the majority of Spec, BioBench and Parsec applications even with 4 KB pages (35 out of 47 applications). The reason is that these suites consist of several benchmarks with

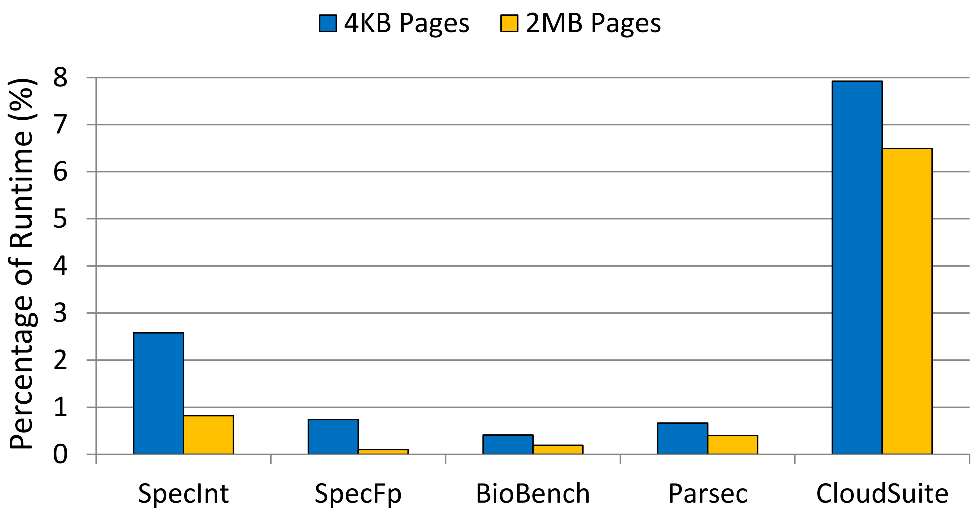


Figure 3.6: Comparison of the MMU performance with other benchmark suites (geometric mean per suite).

small working sets that fit in the TLB hierarchy. Although there are some benchmarks that cause high MMU overheads (e.g. *mcf*, *omnetpp*, *cactusADM*, *mummer*, *tigr*, *canneal*) they still have smaller working sets, exhibit less kernel activity, and enjoy better performance improvement with huge pages compared to the scale-out applications. Based on these findings, we corroborate a previous study [50] that pointed out the distinct characteristics of scale-out applications compared to other benchmarks, and we further show that the same observation holds with respect to the MMU behavior.

Findings

- Scale-out applications stress more the MMU performance compared to other compute-intensive and multi-threaded applications.

3.4.7 Interference in the cache hierarchy

In this section we quantify the interference in the data-cache hierarchy between the application data and the page table references. To accomplish this, we count the number of L1, L2, and LLC misses for the two page-size configurations. Figure 3.7 shows the percentage of reduced cache misses due to changing the page-size from 4 KB to 2 MB, and Figure 3.8 shows the number of cache misses (L1, L2 and LLC) per 1000 instructions with 4 KB and 2 MB pages.

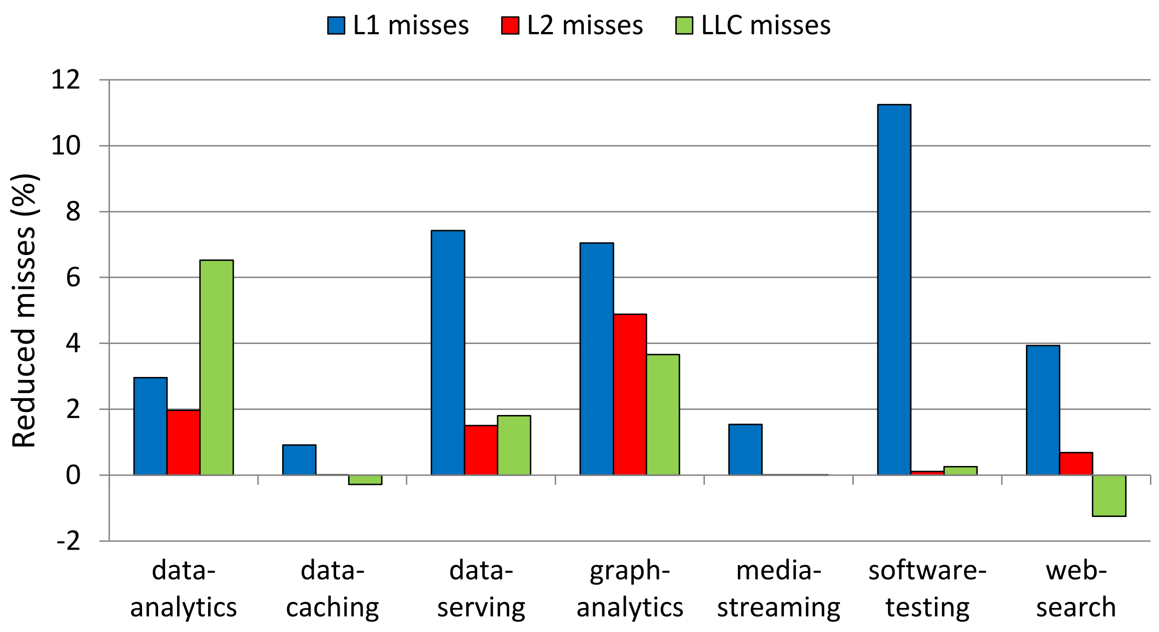


Figure 3.7: Percentage of reduced L1, L2 and Last Level Cache (LLC) misses due to increasing the page-size from 4 KB to 2 MB. The results show that by improving the MMU performance, less interference occurs in the cache hierarchy between the application data and the page table.

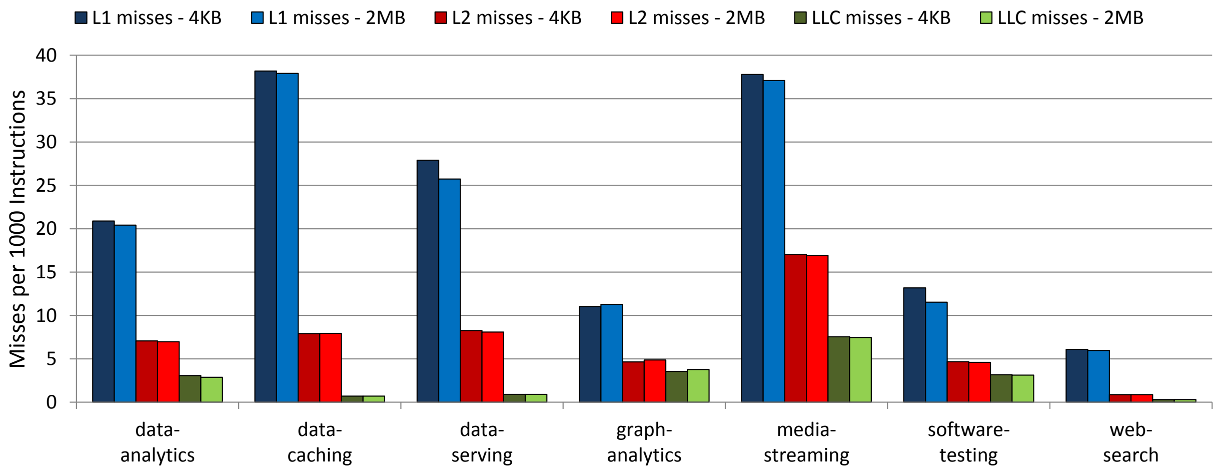


Figure 3.8: Cache misses (L1, L2 and LLC) per 1000 instructions with 4 KB and 2 MB pages.

We observe that the number of cache misses for most scale-out applications reduces by up to 11.2% for L1 cache (*software-testing*), 4.8% for L2 cache (*graph-analytics*) and 6.5% for LLC (*data-analytics*). This happens due to the improved MMU performance that causes fewer memory accesses due to page walks. In addition, improved MMU performance reduces the interference between the application data and the page table in the cache hierarchy because: (i) the page table occupies less memory space due to the elimination of one level for those memory regions that are mapped by 2 MB pages, as explained in Section 2.3.4, and (ii) page walks occur less often and are cheaper (Figures 3.4 and 3.5). The only exceptions are *data-caching* and *web-search* which suffer more frequently from page walks with 2 MB than with 4 KB pages as we showed earlier in Section 3.4.4, increasing slightly the number of LLC misses.

Moreover, we notice that just reducing the number of L1 misses, as it happens for (*software-testing*), does not affect significantly the performance because the L1 misses can be hidden by the out-of-order execution. However, we observe a correspondence between performance improvement and reduced data-cache interference—in L2 cache and LLC—for those applications that benefit most from 2 MB pages (*data-analytics*, *data-serving*, and *graph-analytics*).

Findings

- Poor MMU performance can result in increased interference between the application data and the page table in the cache hierarchy.

3.4.8 Interaction with Hardware Prefetchers

Previously we showed the interference between the application data and the page table in the cache hierarchy. However, the page table can be cached up to the L1 cache. Here we quantify this interference due to the activity of the hardware prefetchers from the MMU performance point of view.

Our experimental machine has four prefetching mechanisms; two of them (DCU and IP stride) are responsible for prefetching data into the L1 cache, while the other two (ACL and Spatial) are responsible for prefetching data into the L2 and the LLC [63]. Table 3.5 summarizes the results of the MMU performance due to data accesses with 2 MB pages for three different prefetcher configurations: (i) all prefetchers are disabled, (ii) only L1 prefetchers (DCU and IP prefetchers) are enabled, and (iii) all prefetchers are enabled. The table shows the total number of page walks, the average number of cycles per page walk, and the total number of cycles spent in page walks, normalized to the case when all prefetchers are disabled. We make the following observations.

We see that the total number of page walks changes for different prefetcher configurations and actually increases for most scale-out applications compared to when all prefetchers are disabled. These results were not expected since the prefetcher requests are supposed not to trigger page walks [63]. We speculate that the number of page walks differs across configurations because the prefetch requests either affect the TLB replacement policy (positively or negatively depending on the application), or indeed trigger page walks.

Similarly, we observe that the total number of cycles spent in page walks (and the average cost of page walk respectively) changes across the various configurations. More specifically, the number of page walk cycles is lower when all prefetchers are disabled for most of the scale-out applications. These results indicate that the prefetchers fetch aggressively application data that interfere with the page table in the cache hierarchy. However, we see that the hardware prefetchers reduce by 47% the cycles spent in page walks for *graph-analytics*, even though the number of page walks is reduced by only 4%. Thus, we observe an interaction between the hardware prefetchers and the MMU performance that require further research and documentation.

Findings

- The interference between the application data and the page table in the cache hierarchy due to the activity of the hardware prefetchers suggest that there is potential for reducing the MMU overhead if there is isolation between them in the memory hierarchy.

	ON only L1 prefetchers			ON all prefetchers		
Benchmark	**#page walks**	**#average cycles per page walk**	**#cycles spent in page walks**	**#page walks**	**#average cycles per page walk**	**#cycles spent in page walks**
Data-analytics	1.11	1.49	1.65	1.00	1.28	1.29
Data-caching	0.79	1.08	0.86	0.77	1.17	0.90
Data-serving	1.22	1.02	1.25	1.22	0.98	1.20
Graph-analytics	0.98	0.85	0.83	0.96	0.55	0.53
Media-streaming	1.00	1.05	1.05	1.06	1.07	1.14
Software-testing	0.81	1.02	0.83	1.04	1.14	1.19
Web-search	1.62	0.94	1.53	1.66	0.95	1.57

Table 3.5: Summarized results for page walks, average cycles per page walk and cycles spent in page walks with 2 MB pages for various prefetcher configurations normalized to the case when all prefetchers are disabled (lower is better).

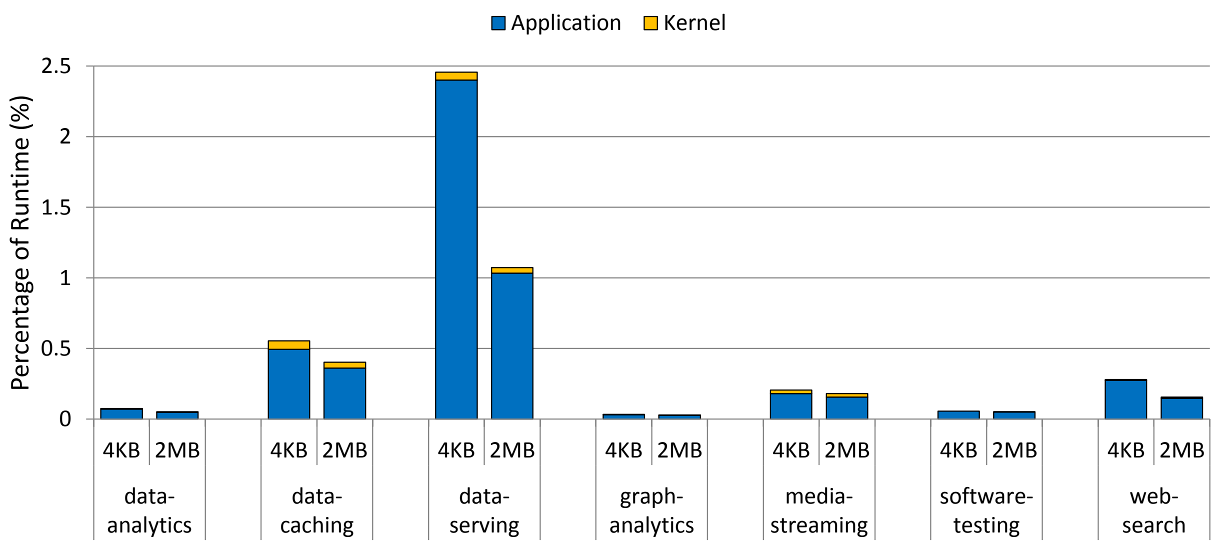


Figure 3.9: Page walk overhead due to instruction TLB misses.

3.4.9 Instruction TLB misses

Scale-out applications have been shown to suffer from high instruction cache-miss rates [50]. Figure 3.9 shows the time spent in TLB misses that trigger page walks due to instruction accesses. We see that all scale-out applications, except for *data-serving*, spend a negligible amount of time in page walks due to instruction accesses (less than 0.6%). However, *data-serving* spends 2.45% of the total execution time in page walks due to instruction accesses with 4 KB pages. This MMU overhead accounts for 25% of the total MMU overhead, and far exceeds typical TLB results for instruction accesses [29]. The reason lies on the software implementation of *data-serving* which is based on a high-level language (Java) with a managed runtime and extensive use of libraries. When 2 MB pages are employed, we find that the MPKI reduces by almost 50%, possibly due to the lower interference between instruction and data entries in the L2 TLB. However, the MPKI of TLB misses due to instruction accesses is still comparable to that of TLB misses due to data accesses (0.8 MPKI for instruction TLB misses vs. 3.7 MPKI for data TLB misses).

Findings

- Instruction references may add non-negligible MMU overheads due to high-level languages and libraries.

3.4.10 Summary & Implications

We have shown that the MMU overhead for the scale-out applications is significantly high, up to 16% of the total execution time. As the datasets for these applications constantly grow, these overheads are expected to increase. Thus, improving the MMU performance should be of paramount importance for the efficient execution of such applications in the big-data era [27, 28]. Moreover, to quantify the correlation between the MMU performance and the application performance, we conducted experiments with huge pages. The results show that lower MMU overhead yields up to 13.9% application speedup. However, even though huge pages reduce the MMU overhead, the hardware support for huge pages is limited and may actually harm the performance. These findings point to the need to improve the limited hardware support for huge pages. Furthermore, kernel code contributes significantly to the total MMU overhead for most of the scale-out applications mainly due to intense network and file-system activities. This behavior requires further investigation from both operating system and architecture researchers. Finally, the interference between application data and the poor MMU performance indicates the need for an holistic approach in order to boost the performance of the memory hierarchy.

3.5 Potential Improvements in the MMU

In this section we discuss potential solutions, and we present upper-bound analyses of performance improvements in the MMU.

3.5.1 Virtual Caches

Virtual caches [26, 36, 37, 80, 100, 114, 116] have been proposed as an alternative to reduce the performance and power dissipation overheads of the MMU. Virtual caches use virtual addresses to access the cache hierarchy down to a certain level and only consult the TLB on a cache miss beyond the supported level in the cache hierarchy. Although virtual caches provide attractive properties, ensuring correct execution requires extra hardware support and complexity (due to synonyms, homonyms, protection access rights, etc.).

We want to investigate the potential of virtual caches for scale-out applications regarding performance, assuming two virtual cache designs. The first design uses virtual

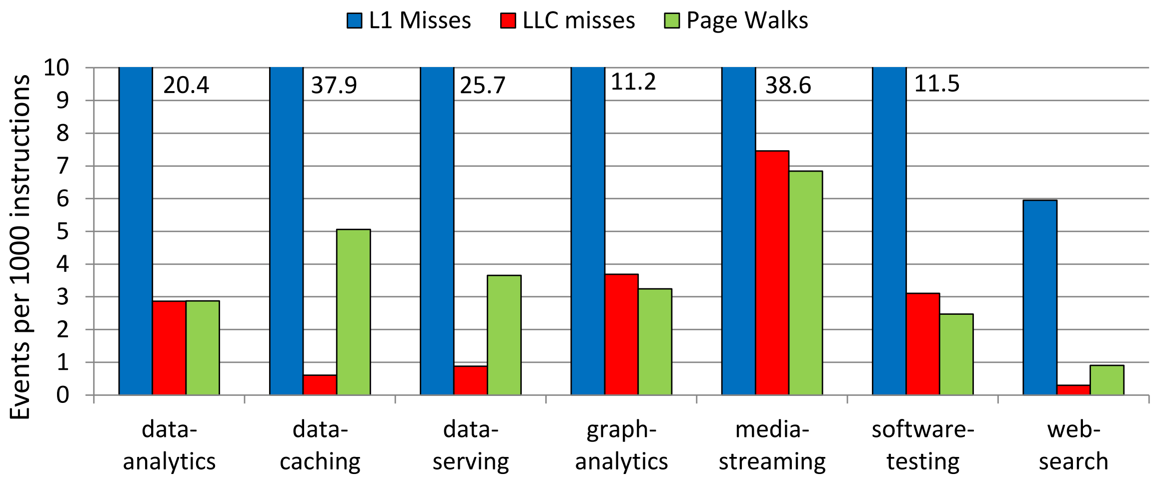


Figure 3.10: Comparison of TLB misses that trigger page walks and LLC misses per 1000 instructions due to data accesses with 2 MB pages.

addresses only for accessing the L1 cache and looks up the MMU on L1 misses. The second design uses virtual addresses for accessing the whole cache hierarchy (L1, L2, and last-level cache), and looks up the MMU on LLC misses. To this end, we compare the MPKI of TLB misses that trigger page walks (green bar) with the MPKI of L1 cache (blue bar) and LLC (red bar) with 2 MB pages in Figure 3.10. We make the following observations.

First, the results show that all applications experience significantly higher L1 miss rate compared to the page walks rate. In addition, four out of seven applications experience similar or higher LLC miss rate compared to the page walks rate. The reason is that these scale-out applications operate on large datasets suffering from LLC misses that are spread over a big memory space that will likely miss in the TLB as well. Consequently, although virtual caches would help in reducing the power dissipated in the TLB hierarchy, they would provide similar behavior in terms of performance due to the exposed cost of address translation in cache misses. Second, we observe that *data-caching*, *data-serving*, and *web-search* suffer more often from page walks than from LLC misses. Such behavior indicates that there is useful data in the cache hierarchy that is not covered by the TLBs, thus exposing the inadequate design of current MMUs.

Based on these findings, we conclude that employing virtual caches and removing the MMU from the critical path would bring negligible performance improvement while it would add significant complexity in the implementation of the system.

3.5.2 Perfect MMU Caches

The MMU cache helps in reducing the cost of page walks by caching memory references of the upper levels of the page table. In this part of our analysis we evaluate the potential for reducing the time spent in page walks by implementing a perfect MMU cache. This means that the page walk requires only one memory reference that always hits in some level of the cache hierarchy. Bhattacharjee [28] performed a similar analysis to show potential improvements due to perfect MMU caches. We go one step further and quantify also the impact of the hit-level in the cache hierarchy during the page walk for perfect MMU caches.

Using microbenchmarks [19], we found that the minimum cost for resolving a page walk that completely hits in the MMU cache and requires only one page table reference that hits in the L1, L2, and LLC cache is 12, 20, and 43 cycles on our platform, respectively. Based on these values per page walk and the actually measured number of page walks with performance counters, we estimate potential performance improvements of perfect MMU caches. Note that we assume pessimistically that the TLB misses have no effect on the rest of the execution pipeline, so that the baseline remains the same for all analyses.

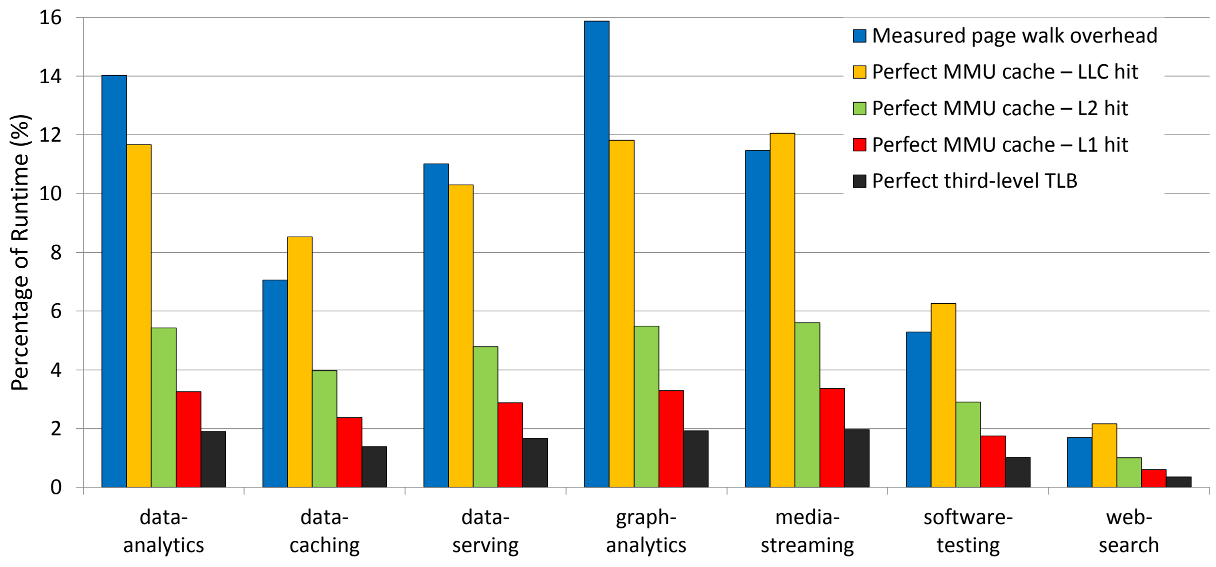
$$PerfectMMU_{LLC}(\%) = \frac{TLB\_Misses * 43cycles}{Total\_Execution\_Time} \tag{3.2}$$

$$PerfectMMU_{L2}(\%) = \frac{TLB\_Misses * 20cycles}{Total\_Execution\_Time} \tag{3.3}$$

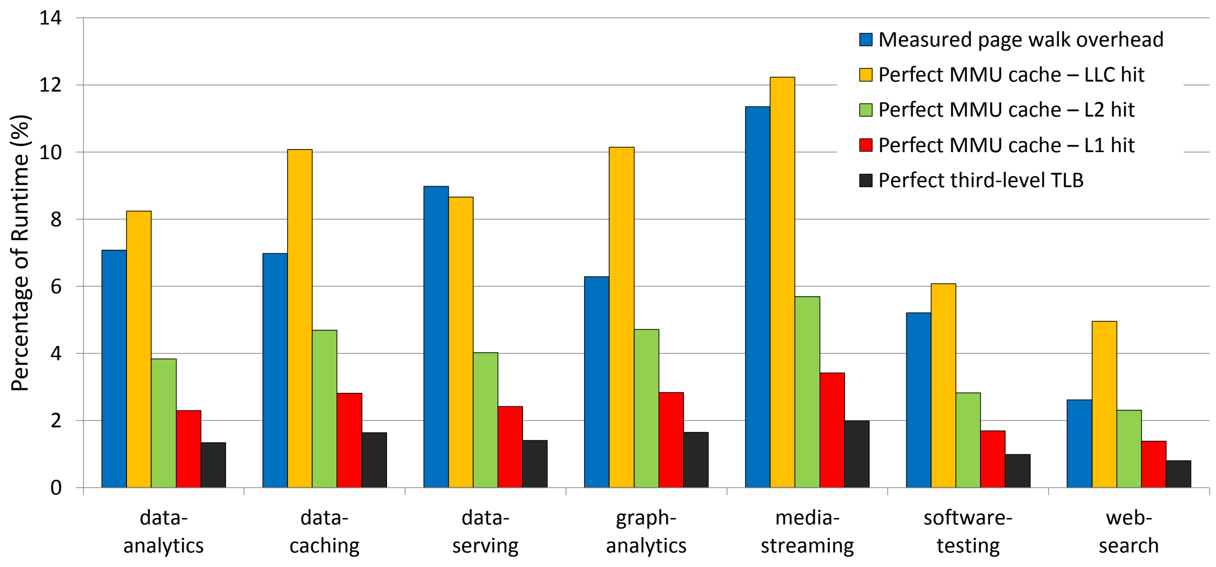
$$PerfectMMU_{L1}(\%) = \frac{TLB\_Misses * 12cycles}{Total\_Execution\_Time} \tag{3.4}$$

In Figure 3.11 we plot the percentage of time spent in page walks due to data accesses (i) for the real evaluated hardware (blue bar), (ii) enhanced with perfect MMU caches that require a single memory reference that hits in the LLC (PerfectMMU$_{LLC}$ – Equation 3.2 – yellow bar), (iii) in the L2 cache (PerfectMMU$_{L2}$ – Equation 3.3 – green bar), and (iv) in the L1 cache (PerfectMMU$_{L1}$ – Equation 3.4 – red bar).

We observe that for 4 KB pages, the actual measured performance overhead is close to that of PerfectMMU$_{LLC}$ or even lower. Since the MMU cache is not perfect for the real measurements, we conclude that the page table references with 4 KB pages typically hit earlier in the cache hierarchy, well before accessing the LLC. Regarding the configuration with 2 MB pages, we observe that the measured performance overhead is lower than that



(a) 4 KB Pages



(b) 2 MB Pages

Figure 3.11: Upper-bound results for potential improvements in the MMU organization.

of PerfectMMU$_{LLC}$ and close to that of the PerfectMMU$_{L2}$. This implies that the page walks with 2 MB pages typically hit in L2.

Regarding the potential improvements of the perfect MMU cache itself, that motivated also a recent proposal for improving their performance [28], we notice that the perfect MMU cache brings better performance improvement for the scale-out applications with 4 KB pages than with 2 MB pages. However, the performance benefits still depend heavily on the level of the cache hierarchy where the page walk hits, indicating the need to keep the page table references as close as possible to the processor.

3.5.3 Perfect Cache Interference

Overall, the perfect MMU cache provides limited performance benefits for the scale-out applications unless it is incorporated with a mechanism that preserves or promotes the page table references in the cache hierarchy closer to the processor (Figure 3.11). On the other hand, we showed that interference exists between the application data and the page table in the cache hierarchy, increasing the average cost of page walks and degrading the application performance.

The interference between application data and page table references in the cache hierarchy has been pointed by Wu et al. [117]. However, their study targeted compute-intensive applications that exhibit high cache hit ratio and low TLB miss ratio. Thus, their proposal treated the page table references as polluting cache entries by de-prioritizing them through the cache replacement policy in favor of application data. We believe that an opposite approach should be considered in the context of scale-out applications that suffer from poor data-cache locality, so that the page walks hit early in the cache hierarchy. Such a research direction is similar in vein with [50] that advocated for preserving instruction references in the cache hierarchy due to the high number of expensive instruction cache misses that take place in scale-out applications.

3.5.4 Perfect TLBs

Here we discuss the possibility of employing a *perfect third-level* TLB that always hits (PerfectTLB – Equation 3.5 – black bar in Figure 3.11), i.e., the TLB has unlimited entries or reach, having the same latency (7 cycles) as the actual L2 TLB [63]. The results show that in this case, the page walk overhead is reduced to less than 2% for all scale-out applications making the use of virtual memory almost free in terms of performance.

$$Perfect_{TLB}(\%) = \frac{TLB\_Misses * 7cycles}{Total\_Execution\_Time} \tag{3.5}$$

One direction for achieving such performance is through architecting a third-level TLB with a high number of entries. However, such an implementation is likely unfeasible according to CMOS technology predictions [7] due to leakage power and area overheads. On the other hand, novel memory technologies (e.g. STT-RAMs, Memristors, NEMs) have been proposed to overcome these CMOS' limitations for other on-chip components such as last-level caches [40, 118]. Leveraging their unique characteristics, i.e., non-volatility, low static power, and high area density, and designing novel third-level TLBs should be considered for future research in our opinion.

Another future direction for improving the MMU performance is to design a third-level *range TLB* that can capture efficiently ranges of pages without constraints on the coverage by a single TLB entry (in contrast to [96, 97, 111]). Direct Segments [27] actually follows this approach, but they provide only a single range. Taking advantage of the fact that a third-level range TLB would not be on the critical path of every memory operation but would be accessed only when misses occur in the higher TLB levels, a more complex design with higher latency and improved range capacity could be beneficial.

Finally, TLB misses could be effectively hidden through smart prefetching. Surprisingly we notice limited proposals in the literature for TLB prefetching [30, 76, 104]. We believe that the high overheads of the MMU for scale-out applications in combination with their distinct characteristics—low locality and limited sharing among threads [50]—deserves an effort in optimizing prefetching TLB entries, similarly as inter-core cooperative prefetching [89] leveraged the frequent sharing patterns of the multi-threaded applications to boost TLB performance.

3.6 Related Work

The MMU performance has attracted the interest of both academia and industry for several decades. Early evaluations of the MMU showed its importance in the overall processor performance [17, 39, 42, 93, 103]. However, these studies were conducted under systems with limited physical memory, less sophisticated MMU organizations, and different workloads, compared to today's trends in the MMU architectural support and the big-memory scale-out applications, respectively.

Jacob and Mudge [68] compared various MMU organizations and showed that the total MMU overhead is roughly twice to what was previously thought due to the interference between application data and page table in the cache hierarchy. Kandiraju et al. [75] presented a detailed characterization of the data TLB behavior for the Spec2000 benchmark suite. The authors suggested that multi-level TLBs are useful in cutting down access times and evaluated different kinds of prefetching. McCurdy et al. [90] evaluated the MMU performance under scientific applications, addressing the limited TLB support for 2 MB pages and concluded that a wrong choice of page size can result in performance degradations of up to nearly 50%, while Morari et al. [92] evaluated the TLB miss impact in future HPC systems.

The most recent work in characterizing and analyzing the TLB performance was conducted in the context of the Parsec multi-threaded applications [29]. The authors showed that TLB misses are predictable due to sharing patterns among threads, and that inter-core TLB cooperation and prefetching mechanisms can improve TLB performance.

Finally, Basu et al. [27] showed that big-memory applications stress the MMU performance even with 1G pages. However, their study includes a subset of the applications we use in this chapter. Similarly, Bhattacharjee [28] showed that a significant amount of execution time is due to MMU overhead. However, their study did not focus on scale-out applications.

In contrast to previous works, we comprehensively characterize the MMU performance using representative scale-out applications from CloudSuite. We also provide deep insights in the interactions between the MMU and other processor components, and we point out to directions for improving the MMU performance in the context of emerging memory-intensive scale-out applications.

3.7 Summary

Understanding the characteristics of scale-out applications and identifying performance inefficiencies has turned out to be a fundamental requirement to boost the efficiency of datacenters and to further spread the deployment of the cloud-computing paradigm.

With this goal in mind, we comprehensively analyzed the performance execution of the Memory Management Unit under the execution of scale-out applications. We showed that the MMU overhead accounts for up to 16% of the total execution time and we quantified the interference between application data and page walks. By reducing the MMU overheads through huge pages, we found that the application performance accelerates by up to 13.9% due to better exploitation of the available execution resources. However, the limited hardware support for huge pages may harm performance.

Consequently, based on upper-bound analyses for perfect improvements in the MMU, we suggested directions for improving the MMU performance. In the rest of this thesis, we further pursuit the suggested concept of *range TLB* to improve the performance (Chapter 4) and the energy-efficiency (Chapter 5) of virtual memory.

4 Fast Address Translation with Ranges

4.1 Introduction

Recent studies and this thesis show that modern workloads can experience high execution-time overheads, up to 50%, due to page table walks [27, 28, 77]. The root cause is the *limited TLB reach*; because TLB address translation is on the processors' critical path, it requires low access times which constrain TLB size and thus the number of pages that experience this access time. This overhead is likely to grow, because physical memory sizes are still growing. Furthermore, many modern applications have an insatiable desire for memory—they increase their data set sizes to consume all available memory for each new generation of hardware [27, 50].

Previous research has focused on increasing the TLB reach and improving the performance of paging in the following three ways.

1. Multipage mappings use one TLB entry to map multiple pages (e.g., 8-16 pages per entry) [96, 97, 111]. Mapping multiple pages per entry increases TLB reach by a

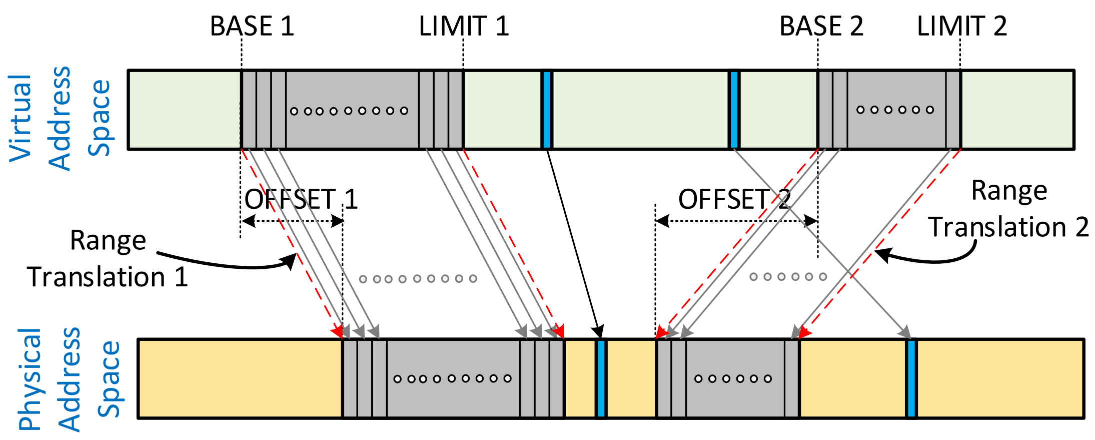


Figure 4.1: RMM introduces the key concept of range translations: an efficient representation of contiguous virtual pages mapped to contiguous physical pages, complementary to paging. Each range translation uses BASE, LIMIT, and OFFSET values to perform translation of an arbitrary sized range. The figure shows the virtual-to-physical mappings of an application with RMM; all the mappings use page-based translation, but some of them use also range-based translation redundantly when enough contiguity is present.

small fixed amount, but has alignment restrictions, and still leaves TLB reach far below modern gigabyte-to-terabyte physical memory sizes.

2. Huge pages map much larger fixed size regions of memory, on the orders of 2 MB to 1 GB on x86-64 architectures. Use of huge pages (THP [5] and libhugetlbfs [8]) increase TLB reach substantially, but also suffer from size and alignment restrictions and still have limited reach.

3. Direct segments provide a single arbitrarily large segment per process and standard paging for the remaining virtual address space [27, 52]. For applications that can allocate and use a single segment for the majority of their memory accesses, direct segments eliminate most of the paging cost. However, direct segments only support a single segment and require that application writers explicitly allocate a segment during startup.

The goal of our work is to provide a robust virtual memory implementation with near zero overheads that is transparent to applications, enables fast address translation with no alignment restrictions, and retains all the benefits of paging across a variety of workloads.

We introduce Redundant Memory Mappings (RMM), a novel hardware/software co-designed implementation of virtual memory. RMM adds a redundant mapping, in addition to page tables, that provides a more efficient representation of translation information for a range of pages that are both physically and virtually contiguous. RMM exploits the natural contiguity in address space and keeps the complete page table as a fall-back mechanism.

RMM relies on the concept of *range translation*. Each range translation maps a contiguous virtual address range to contiguous physical pages, and uses BASE, LIMIT, and OFFSET values to perform translation of an arbitrary sized range. Range translations are only base-page-aligned and redundant to paging; the page table still maps the entire virtual address space. Figure 4.1 illustrates an application with two ranges mapped redundantly with paging as well as range translations.

Analogous to paging, we add a software managed *range table* to map virtual ranges to physical ranges and a hardware *range TLB* in parallel with the last-level page TLB, e.g., the L2-page TLB, to accelerate their address translation. Because range tables are redundant to page tables, RMM offers all the flexibility of paging and the operating system may use or revert solely to paging when necessary.

To increase contiguity in range translations, we extend the OS's default lazy demand page allocation strategy to perform eager paging. Eager paging instantiates pages in physical memory at allocation request time, rather than at first-access time as with demand paging. The resulting OS automatically maps most of process's virtual address space with orders of magnitude fewer ranges than paging with Transparent Huge Pages [5]. On a wide variety of workloads consuming between 350 MB – 75 GB of memory, we find that RMM has the potential to map more than 99% of memory for all workloads with 50 or fewer range translations (see Table 4.2 in Section 4.3).

To evaluate this design, we implement RMM software support in Linux kernel v3.15.5. We emulate the hardware using a combination of hardware performance counters from an x86 execution and functional TLB simulation in BadgerTrap [51]—the same methodology as in prior TLB studies [27, 28, 52]. We compare RMM to standard paging, Clustered TLBs, huge (2 MB and 1 GB) pages, and direct segments (one range per program). RMM robustly performs substantially better than the former three alternatives on various workloads, and almost as fast as Direct segments when one range is applicable. However with RMM, more applications enjoy reductions in translation overhead without programmer intervention. Overall, RMM reduces the overhead of virtual memory to less than 1% on average.

In summary, the main contributions of this chapter are:

- We show that diverse workloads exhibit an abundance of contiguity in their virtual address space.
- We propose Redundant Memory Mappings, a hardware/ software co-design, which includes a fast and redundant translation mechanism for ranges of contiguous virtual pages mapped to contiguous physical pages, including operating system support to detect and manage ranges.
- We prototype RMM in Linux and evaluate it on a broad range of workloads. Our results show that a modest number of ranges map most of memory. Consequently, the range TLB achieves extremely high hit rates, eliminating the vast majority of costly page walks compared to virtual memory systems that use paging alone.

The rest of the chapter is organized as follows: Section 4.2 provides background information on some important previous work; Section 4.3 provides an overview of RMM showing the key opportunity that exploits; Sections 4.4 and 4.5 describe the architectural support and the OS support respectively; Section 4.6 discusses various implementation details; Section 4.7 describes the methodology used to evaluate our design; Section 4.8 presents the results; Section 4.9 discusses the related work, and Section 4.10 concludes our study.

4.2 Background

This section and Table 4.1 overview the closely related approaches to reducing paging overheads and compare them to RMM. Section 4.9 discusses related work more generally.

Multipage Mapping approaches, such as sub-blocked TLBs [111], CoLT [96] and Clustered TLBs [97], pack multiple Page Table Entries (PTEs) into a single TLB entry. These designs leverage *default* OS memory allocators that either (i) assign small blocks of contiguous physical pages to contiguous virtual pages (Sub-blocked TLBs and CoLT), or (ii) map a small set of contiguous virtual pages to clustered sets of physical pages (Clustered TLB). However, they pack only a small multiple of translations (e.g., 8-16) per entry, which limits their potential to reduce page walks for large working sets.

	Transparent to Application	Kernel support	Hardware support	# of entries	Maximum reach per entry	Application domain	No size-alignment restrictions
Multipage Mappings [96, 97, 111]	✓	✗	✓	512	32 KB to 16 MB	any	✗
Transparent Huge Pages [5, 94]	✓	✓	✓	32	2 MB	any	✗
libhugetlbfs [8]	✗	✓	✓	4	1 GB	big memory	✗
Direct segments [27]	✗	✓	✓	1	unlimited	big memory	✓
Redundant Memory Mappings	✓	✓	✓	N	unlimited	any	✓

Table 4.1: Comparison of Redundant Memory Mappings with previous approaches for reducing virtual memory overhead.

Huge Pages using Transparent Huge Pages (THP) [5] and libhugetlbfs [8] increase the TLB reach by mapping very large regions with a single entry. The x86-64 architecture supports mixing 4 KB with 2 MB and 1 GB pages, while other architectures support more sizes [91, 101, 107]. The effectiveness of huge pages is limited by the size-alignment requirement: huge pages must have size-aligned physical addresses, and thus the OS can only allocate them when the available memory is size-aligned and contiguous [96, 97]. In addition, many commodity processors provide limited numbers of huge page TLB entries, which further limits their benefit [27, 52, 77].

Direct segments [27] are a hardware/software approach that map a *single unlimited* range of contiguous virtual memory to contiguous physical memory using a single hardware segment, while the rest of the virtual address space uses standard paging. A virtual address is mapped by a direct segment or paging, but never both. Direct segments introduce BASE, LIMIT, and OFFSET registers to eliminate the page walks within the segment. However, the mechanism requires that (i) applications explicitly allocate a direct segment during startup, and (ii) the OS can reserve a single large contiguous range of physical memory for a segment. Thus, direct segments are only suitable for big-memory workloads and require application changes.

Table 4.1 summarizes the characteristics of these approaches and compares them to RMM. RMM is completely transparent to applications and maps multiple ranges with no size-alignment restrictions, where each range contains an unrestricted amount of memory.

	Pages	Ideal RMM ranges		
Benchmark	**4 KB + 2 MB**	**total**	**99% coverage**	**largest**
astar	5129 + 158	55	7	76.2%
mcf	1737 + 839	55	1	99.0%
omnetpp	2041 + 77	54	12	60.2%
cactusADM	1365 + 333	112	49	2.4%
GemsFDTD	3117 + 414	73	6	71.7%
soplex	4221 + 411	61	5	41.9%
canneal	10016 + 359	77	4	90.9%
streamcluster	1679 + 55	78	14	83.8%
mummer	29571 + 172	17	4	57.5%
tigr	28299 + 235	16	3	97.9%
Graph500	8983 + 35725	86	3	50.4%
Memcached	4243 + 36356	82	2	98.6%
NPB:CG	2540 + 26058	84	5	28.8%
GUPS	2210 + 32803	92	1	99.7%

Table 4.2: Total translation entries mapping the application's memory with: (i) Transparent Huge Pages of 4 KB and 2 MB pages [5] and (ii) ideal RMM ranges of contiguous virtual pages to contiguous physical pages. (iii) Number of ranges that map 99% of the application's memory, and (iv) percentage of application memory mapped by the single largest range.

4.3 Redundant Memory Mappings

We observe that many applications naturally exhibit an abundance of contiguity in their virtual address space and the number of ranges needed to represent this contiguity is low.

Abundance of address contiguity. We quantify address contiguity by executing applications on x86-64 hardware (see Section 4.7 for workload and methodology details), and periodically scan the page table, measuring the size of virtual address ranges where all pages are mapped with the same permissions. Table 4.2 shows the minimum number of ranges of contiguous virtual pages that the OS could map to contiguous physical pages. The workloads require between 16 to 112 ranges to map their entire virtual address space. However, the number of ranges to cover 99% of the application's memory space falls to less than 50. Although a single range maps 90% or more of the virtual memory for 5 of the 14 workloads, the rest require multiple ranges. These results suggest that a small number of range translations have the potential to efficiently perform address translation for the majority of virtual memory addresses.

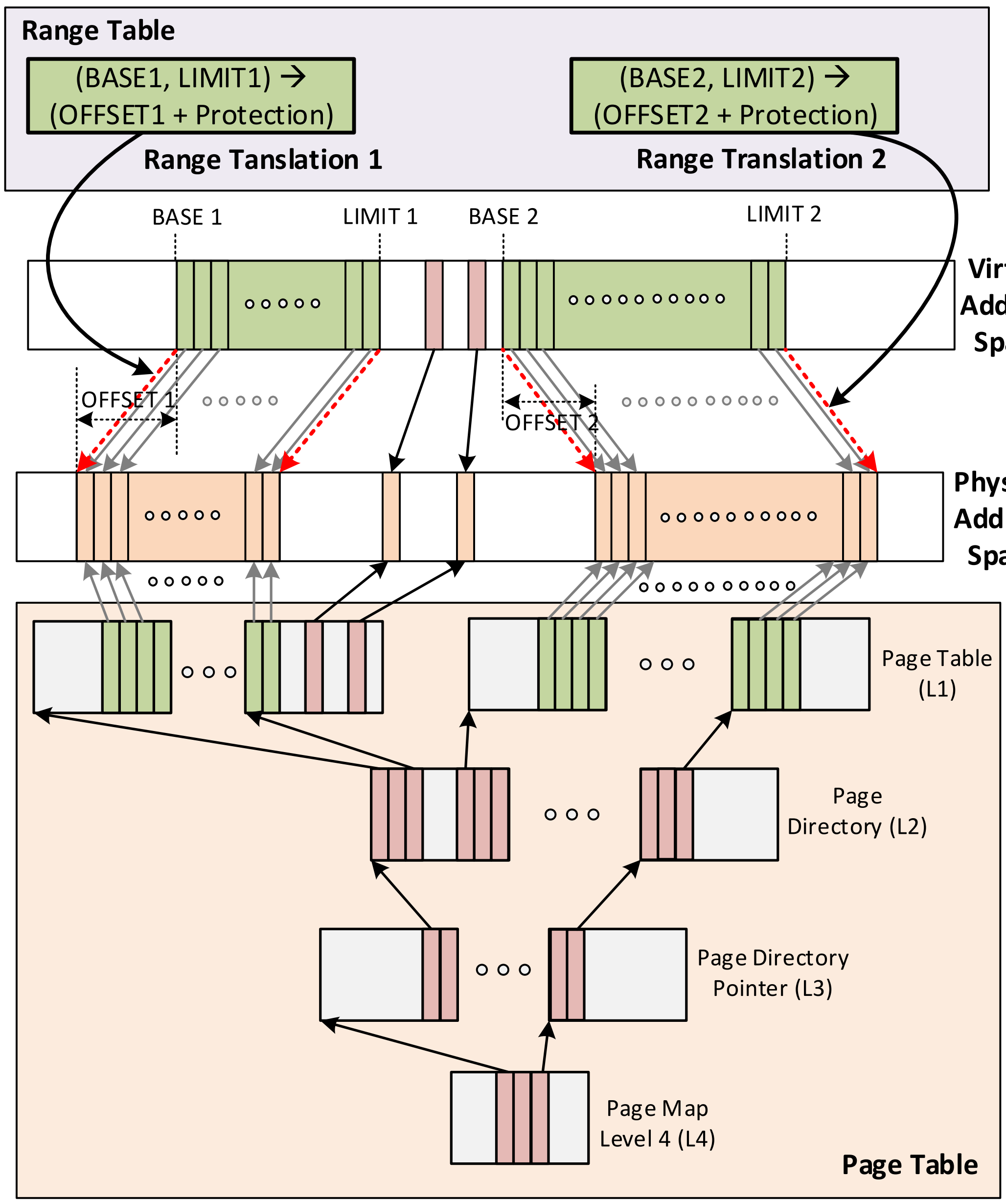


Figure 4.2: Redundant Memory Mappings design. The application's memory space is represented *redundantly* by both pages and range translations. The page table holds all virtual-to-physical mappings at page granularity for a process. The range table of RMM (not shown) holds mappings for most of the process's address space redundantly with range translations. Each range translation uses BASE_i, LIMIT_i, and OFFSET_i values to perform translation of an arbitrary sized range. The BASE and LIMIT values denote the range in the virtual address space. The OFFSET value holds the beginning of the range in the physical address space minus BASE, and the protection bits. RMM performs translation from the virtual to physical address space for a *virtual address* that falls inside a range translation ($\text{BASE}_i \leq$ virtual address $< \text{LIMIT}_i$) by adding the OFFSET value to the virtual address, i.e., physical address = virtual address + OFFSET_i.

	Page Translation (x86-64)	+	Range Translation
Architecture	TLB page table CR3 register page table walker		range TLB range table CR-RT register range table walker
OS	page table management demand paging		range table management eager paging

Table 4.3: Overview of Redundant Memory Mappings.

4.3.1 Overview

The above measurements motivate the RMM approach. (i) The OS uses best-effort allocation to detect and map contiguous virtual pages to contiguous physical pages in a range table in addition to mapping with the page table. (ii) The hardware range TLB caches multiple range translations providing an alternate translation mechanism, parallel to paging. (iii) Most addresses fall in ranges and hit in the range TLB, but if needed, the system can revert to the flexibility and reduced fragmentation benefits of paging.

Definition: A *range translation* is a mapping between contiguous virtual pages mapped to contiguous physical pages with uniform protection bits (e.g., read/write). A range translation is of unlimited size and base-page-aligned. A range translation is identified by BASE and LIMIT addresses, and performs address translation with the OFFSET value and the protection access rights. The BASE and the LIMIT hold the beginning and the end of the range translation in the virtual address space. The OFFSET value holds the beginning of the range translation in the physical address space minus the BASE value. Thus, to translate a virtual range address to physical address, the hardware adds the virtual address to the OFFSET of the corresponding range, i.e., physical address = virtual address + OFFSET, and uses the range's protection access rights. Figure 4.2 shows how RMM maps parts of the process's address space with both range translations and pages.

Redundant Memory Mappings (RMM) use *range translations* to perform address translation much more efficiently than paging for large regions of contiguous physical addresses. We introduce three novel components to manage ranges: (i) *range TLBs*, (ii) *range tables*, and (iii) *eager paging* allocation. Table 4.3 summarizes these new components and their relationship to paging. The *range TLB* hardware stores range translations and is accessed in parallel to the last-level page TLB (e.g., L2 TLB). The address translation hardware accesses the range and page TLBs in parallel after a miss at the previous-level TLB (e.g., L1 TLB). If the request hits in the range TLB or in the page TLB, the hardware installs a 4 KB TLB entry in the previous-level TLB, and execution continues. In the uncommon case that a request misses in both range TLB and page TLB, and the address maps to a range translation, the hardware fetches the page table entry to resume execution and optionally fetches a range table entry in the background.

RMM performance depends on the range TLB achieving a high hit ratio with few entries. To maximize the size of each range, RMM extends the OS page allocator to improve contiguity with an *eager paging* mechanism that instantiates a contiguous range of physical pages at allocation time, rather than the on-demand default, which instantiates pages in physical memory upon first access. The OS always updates both the page table and the range table to consistently manage the entire memory at both the page and range granularity.

4.4 Architectural Support

The RMM hardware primarily consists of the range TLB, which holds multiple range translations, each of which translates for an unlimited-size range. We describe RMM as an extension to the x86-64 architecture, but the design applies to other architectures as well.

4.4.1 Range TLB

The range TLB is a hardware cache that holds multiple range translations. Each entry maps an unlimited range of contiguous virtual pages to contiguous physical pages. The range TLB is accessed in parallel with the last-level page TLB (e.g., the L2 TLB) and in case of hit, it generates the corresponding 4 KB entry in the previous-level page TLB (e.g., the L1 TLB).

We design the range TLB as a fully associative structure, because each range can be any size making standard indexing for set-associative structure hard. The right side of Figure 4.3 illustrates the range TLB and its logic with N (e.g., 32) entries. Each range TLB entry consists of a *virtual range* and *translation*. The virtual range stores the BASE_i

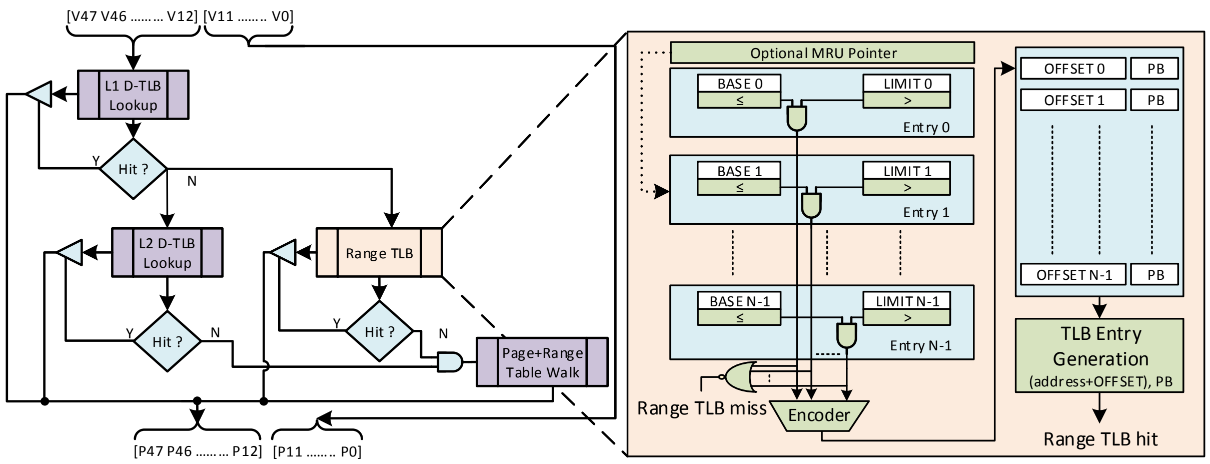


Figure 4.3: RMM hardware support consists primarily of a range TLB that is accessed in parallel with the last-level page TLB.

and LIMIT$_i$ of the virtual address range map. The translation stores the OFFSET$_i$ that holds the start of the range in physical memory minus BASE$_i$, and the protection bits (PB). Additionally, each range TLB entry includes two comparators for lookup operations. Note that we use such OFFSET value (instead of simply using the beginning of the range in the physical address space) because it performs virtual-to-physical address translation with a simple add operation (instead of finding first the distance from the beginning of the range in the virtual address space and then adding that value to the beginning value of the range in the physical address space).

Figure 4.3 illustrates accessing the range TLB in parallel with the L2 TLB, after a miss at the L1 TLB. The hardware compares the *virtual page number* that misses in the L1 TLB, testing BASE$_i$ $\leq$ virtual page number $<$ LIMIT$_i$ for all ranges in parallel in the range TLB. On a hit, the range TLB returns the OFFSET$_i$ and protection bits for the corresponding range translation and calculates the corresponding page table entry for the L1 TLB, as explained in Section 4.3.1. The hardware adds the requested virtual page number to the hit *OFFSET$_i$* value to produce the physical page number and copies the protection bits from the range translation. On a miss, the hardware fetches the corresponding range translation—if it exists—from the range table. We explain this operation in Section 4.4.3 after discussing the range table in more detail.

The range TLB is accessed in parallel with the last-level page TLB and must return the lookup result (hit/miss) within the TLB access latency, which for the L2 TLB on recent

Intel processors is ∼7 cycles [63]. Unlike a page TLB, the range TLB is similar to N fully-associative copies of direct segment's base/limit/offset logic [27] or a simplified version of the range cache [113]: it performs two comparisons per entry instead of a single equality test. Our design can achieve this performance because the range TLB contains only a few entries and it can use fast comparison circuits [81]. Our results in Section 4.8 show that a 32-entry fully-associative range TLB eliminates more than 99% of the page walks for most of our applications, at lower power and area cost than simply increasing the size of the corresponding L2 TLB.

Note that our approach of accessing the range TLB in parallel to the last-level page TLB is because in this chapter we target eliminating the performance overhead of virtual memory due to page walks, i.e., due to L2 TLB misses. Note that our approach of accessing the range TLB in parallel to the last-level page TLB can be extended to the other translation levels closer to the processor, e.g., in parallel to the L1 TLB. Actually, in Chapter 5, we extend the RMM design with an L1-range TLB, accessed in parallel to the L1 TLB, and other mechanisms to improve primarily the energy-efficiency of address translation and eliminate secondarily the performance overhead of virtual memory due to L1 TLB misses.

Optimization. To reduce the dynamic energy cost of the fully associative lookups, we introduce an optional *MRU Pointer* that stores the most-recently-used range translation and thus reduces associative searches of the range TLB. The range TLB first checks the MRU Pointer and in case of a hit, skips the other entries. Otherwise, the range TLB checks all valid entries in parallel. Note that the MRU Pointer can serve translation requests faster than the corresponding page TLB and may further boost performance.

4.4.2 Range table

The range table is an *architecturally visible* per-process data structure that stores the process's range translations in memory. The role of the range table is similar to that of the page table. A hardware walker loads range translations from the range table on a range TLB miss, and the OS manages range table entries based on the application's memory management operations.

We propose using a B-Tree data structure with (BASE_i, LIMIT_i) as keys and OFFSET_i and protection bits as values to store the range table. B-trees are cache friendly and keep

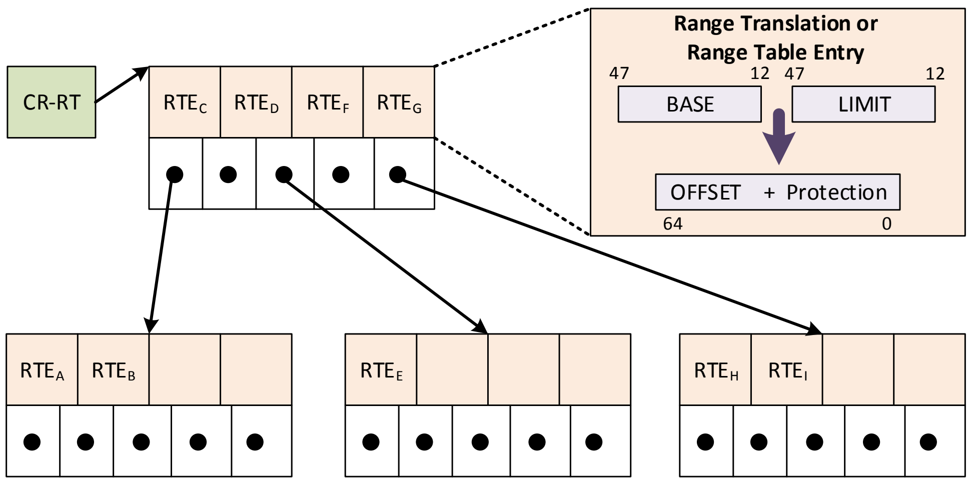


Figure 4.4: The range table stores the range translations for a process in memory. The OS manages the range table entries based on the applications memory management operations.

the data sorted to perform search and update operations in logarithmic time. Since a single B-Tree node may have multiple ranges and children, it is a dense representation of ranges.

The number of ranges per range table node defines the depth of the tree and the average number of node lookups to perform a search/update operation. Figure 4.4 shows how the range translations are stored in the range table and the design of each node. Each node accommodates four range translations and points to five children, e.g., up to 124 range translations in three levels. Since each range translation is represented at page-granularity with the BASE (48 architectural bits −12 bits per page=36 bits), the LIMIT (36 bits), and the OFFSET and protection bits together (64-bits conventional PTE size), thus each range table node fits in two cache-lines. This design ensures the traversal of the range table is cache-friendly, accesses only a few cache lines per operation, and maintains the dense representation. Note that the range table is much smaller than a page table: a *single 4 KB page* stores 128 range translations, which is more than enough for almost all our workloads (Table 4.7). All the pointers to the children are physical addresses, which facilitate walking the range table in hardware.

Analogous to the page table pointer register (CR3 in x86-64), RMM requires a *CR-RT* register to point to the physical address of the range table root to perform address translation, as we will explain next.

4.4.3 Handling misses in the range TLB

On a miss to the range TLB and corresponding page TLB, the hardware must fetch a translation from the memory. Two design issues arise with RMM at this point. First, should address translation hardware use the page table to fetch only the missing PTE or the range table to fetch the range translation? Second, how does the hardware determine if the missing translation is part of a range translation and avoid unnecessary lookups in the range table? Because ranges are redundant, there are several options.

Miss-handling order. RMM first fetches the missing translation from the page table, as all valid pages are guaranteed to be present, and installs it in the previous-level TLB so that the processor can continue executing the pending operation. This choice avoids additional latency from accessing the range table for pages that are not redundantly mapped. *In the background*, the range table walker hardware resolves whether the address falls in a range and if it does, updates the range table with the range table entry. Thus when both the range table and page TLB miss, the miss incurs the cost of a page walk. Any updates to the range TLB occur *off the critical path*.

Identifying valid range translations. To identify whether a miss in the range TLB can be resolved to a range or not, RMM adds a *range bit* to the PTE, which indicates whether a page is part of a range table entry. The page table walker fetches the PTE, and if the range bit is set, accesses the range table in the background. Without this hint, available from redundancy, the range table walker would have to check the range table on every TLB miss. Alternatively, hardware could use prediction to decide whether to access the range table, which requires no changes to page table entries, but we did not evaluate this option.

Walking the range table. Similar to the page table walker, RMM introduces the range table walker that consists of two comparators and a hardware state machine. The range table walker walks the range table in the background starting from the CR-RT register. The walker compares the missing address with the range translations in each range table node and follows the child pointers until it finds the corresponding range translation and installs it in the range TLB. To simplify the hardware, an OS handler could perform the range table lookup.

Shootdown. The OS uses the `INVLPG` instruction to invalidate stale virtual to physical translations (including changes in the protection bits) during the TLB shootdown process [34]. To ensure correct functionality, RMM modifies the `INVLPG` instruction to invalidate all TLB entries and any range TLB entry that contains the corresponding virtual page. The modified OS may thus use this instruction to keep all TLBs and the range TLB coherent through the TLB shootdown process. The OS may also associate each range TLB entry with an address space identifier, similar to TLB entries, to perform context switches without flushing the range TLB.

4.5 Operating System Support

RMM requires modest operating system (OS) modifications. The OS must create and manage range table entries in software and coordinate them with the page table. We modify the OS to increase the size of ranges with an eager paging allocation mechanism. We prototype these changes in Linux, but the design is applicable to other OSes.

4.5.1 Managing range translations

Similar to paging, the process control block in RMM stores a range table pointer (RT pointer) with the physical address of the root node of the range table. When the OS creates a process, it allocates space for the range table and sets the RT pointer. On every context switch, the OS copies the RT pointer to the CR-RT register and then the range table walker uses it to walk the range table.

The OS updates the range table when the application allocates or frees memory or the OS reclaims a page. The OS analyzes the contiguity of the affected page(s). Based on a *contiguity threshold* (e.g., 8 pages), the OS adds, updates, or removes a range translation from the range table. The OS avoids creating small range translations that could cause thrashing in the range TLB. The OS can modify the contiguity threshold dynamically, based on the current number and size of range translations, and the performance of the range TLB (option not explored). The OS updates the range bit in all the corresponding PTEs for the range to keep them consistent.

4.5.2 Contiguous memory allocation

Achieving a high hit ratio in the range TLB and thus low virtual memory overheads requires a small number of very large range translations that satisfy most virtual address translation requests. To this end, RMM modifies the OS memory allocation mechanism to use *eager paging*, which strives to allocate the largest possible range of contiguous virtual pages to contiguous physical pages. Eager paging requires modest changes to Linux's default buddy page allocator.

Default buddy allocator. The buddy allocator splits physical memory in blocks of *orders* using powers-of-two pages, i.e., 2^{order} pages, and manages the blocks using separate *free-lists* per block size. A kernel compile-time parameter defines the *maximum size of memory blocks* (2^{max_order}) and hence the total number of the free-lists. The buddy allocator organizes each free-list in power-of-two blocks and satisfies requests from the free-list of the smallest size. If a block of the desired 2^i size is not available (i.e., free-list[i] is empty), the OS finds the next larger 2^{i+k} size free block, going from $k = 1, 2, \ldots$ until it finds the smallest free block large enough to satisfy the request. The OS then iteratively splits a block in two, until it creates a free block of the desired 2^i size. It then assigns one free block to the allocation and adds any other free blocks it creates to the appropriate free-lists. When the application later frees a 2^i block, the OS examines its corresponding buddy block (identified by its address). If this block is free, the OS coalesces the two blocks, resulting in a 2^{i+1} block. The buddy allocator thus easily splits and merges blocks during allocations and deallocations respectively.

Despite contiguous pages in the buddy heap, in practice most allocations are of a single page because of demand paging. Operating systems use demand paging to reduce allocation latency by deferring page instantiation until the application actually references the page. Therefore, the application's allocation does not trigger OS allocation, but rather when the application first writes or reads a page, the OS allocates a single page (from free-list[0]). Demand allocation at *access-time* degrades contiguity, because (i) it allocates single pages even when large regions of physical memory are available, and because (ii) the OS may assign pages accessed out-of-order to non-contiguous physical pages even though there are contiguous free pages.

Eager paging. Eager paging improves the generation of large range translations by allocating consecutive physical pages to consecutive virtual pages eagerly at allocation, rather than lazily on demand at access time. At *allocation request time* (e.g., when the application performs an mmap, mremap or brk call), if the request is larger than the range threshold, the OS establishes one or more range translations for the entire request and updates the corresponding range and page table entries. We note that demand paging replaced eager paging in early systems. However, one motivation for demand paging was to limit unnecessary swapping in multiprogrammed workloads, which modern large memories make less common [27]. We find that the exponential growth in physical memories and the high cost of TLB misses makes eager paging a better choice with RMM hardware in most cases.

Eager paging increases latency during allocation and may induce fragmentation, because the OS must instantiate all pages in memory, even those the application never uses. However unused memory is not permanently wasted. The OS could monitor memory use in range translations and reclaim ranges and pages with standard paging mechanisms, but we leave this exploration for future work. Allocating memory at request-time generates larger range translations compared to the access-time policy of demand paging and improves the effectiveness of RMM hardware. Note that Section 4.8 quantifies the impact of eager paging on execution time and memory compared to demand paging.

Algorithm. Figure 4.1 shows simplified pseudocode for eager paging. If the application requests an allocation of size N×pages, eager paging allocates the 2^i block, as described above. This simple algorithm only provides contiguity up to the maximum managed block size. If the application requests more memory than the maximum managed block, the OS will allocate multiple maximum blocks. Two optimizations further improve contiguity. First, eager paging could sort the blocks in the free-lists, to coalesce multiple blocks and generate range translations larger than the maximum block. Second, to generate large range translations from allocations that are smaller than the maximum block, eager paging could request a block from a larger size free-list, assign the necessary pages, and return the remaining blocks to the corresponding smaller sized free-lists. These enhancements introduce additional trade-offs that warrant more investigation. Note that in our RMM prototype, we did not implement these two enhancements. Nonetheless, the simple eager paging algorithm generates large range translations for a variety of block sizes and exploits the clustering behavior of the buddy allocator [96, 97].

Finally, eager paging is only effective when memory fragmentation remains low and

ALGORITHM 4.1: RMM memory allocator pseudocode for an allocation request of *number of pages*. When memory fragmentation is low, RMM uses eager paging to allocate pages at *request-time*, creating the largest possible range for the allocation request. Otherwise, RMM uses default demand paging to allocates pages at *access-time*.

```
compute the memory fragmentation;
if memory fragmentation ≤ threshold then
    // low memory fragmentation - use eager paging;
    while number of pages > 0 do
        for (i = MAX_ORDER-1; i ≥ 0; i–) do
            if freelist[i] ≥ 0 and $2^i$ ≤ number of pages then
                allocate block of $2^i$ pages;
                for all $2^i$ pages of the allocated block do
                    construct and set the PTE;
                end
                add the block to the range table;
                number of pages – = $2^i$;
                break;
            end
        end
    end
else
    // high memory fragmentation - use demand paging;
    for (i = 0; i < number of pages; i++) do
        allocate the PTE;
        set the PTE as invalid so that the first access will trigger a page fault and the page will get
        allocated;
    end
end
```

there is ample space to populate ranges at request time. If memory fragmentation or pressure increases, the OS may fall back to its default paging allocation.

4.6 Discussion

This section discusses some of the hardware and operating systems issues that a production implementation should consider, but leaves the implications for automatic and explicit memory management and for applications as future work.

TLB friendly workloads. If an application has small memory footprint and experiences a low page TLB miss rate, the range TLB may provide little performance benefit while increasing the dynamic energy due to range TLB accesses. The OS can monitor the memory footprint and then dynamically enable and disable the range TLB. The OS would still allocate ranges and populate the range table, but then it could selectively enable the range TLB based on performance-counter measurements and workload memory allocation.

Accessed & Dirty bits. The TLB in x86 processors is responsible for setting the *accessed bit* in the corresponding PTE in memory on the first access to a page and the *dirty bit* on the first write. The range TLB does not store per-page accessed/dirty bits for the individual pages that compose a range translation. Thus, on a range TLB hit, the range TLB cannot determine whether it should set the accessed or dirty bit. The OS may address this issue by setting the accessed and dirty bits for all the individual pages of a range translation eagerly at allocation time, instead of at access or write time. If the OS needs to reclaim or swap a page in an active range because of memory pressure, it may. Because the OS manages physical memory at the page-granularity—not at the range granularity—it may reclaim and swap individual pages by dissolving a range completely and then evicting and swapping pages individually. Another option is for the OS to break a range in to multiple smaller ranges and dissolve one of the resulting ranges.

Copy-on-write. Copy-on-write is a virtual memory optimization in which processes initially share pages and the OS only creates separate individual pages when one of the processes modifies the page. This mechanism ensures that these changes are only visible to the owning process and to no other process. To implement this functionality, copy-on-write uses per-page protection bits that trigger a fault when the page is modified. On a fault, the OS copies the page and updates the protection bits in the page table. With RMM, the range translations hold the protection bits at range granularity, not on individual pages. One simple approach is to use range translations for read-only shared ranges, but dissolve a range into pages when a process writes to any of its pages. Alternatively, the OS could copy the entire range translation on a fault. In this thesis we did not evaluate RMM under copy-on-write scenarios, but we consider it as an excellent path for future research.

Fragmentation. Long-running server and desktop systems will execute multiple processes at once and a variety of workload mixes. Frequent memory management requests from complex and short-running workloads may cause physical memory fragmentation and limit the performance of RMM. If the OS cannot find a sufficiently large range of free pages in memory, it should default to paging-only and disable the range TLB. However, abundant memory capacity coupled with fragmentation is not uncommon, since a few pages scattered throughout memory can cause considerable fragmentation [43]. In this case, the OS could perform full compaction [27, 96], or partial compaction with techniques adapted from garbage collection [35, 43].

4.7 Methodology

To evaluate virtual memory system performance on large memory workloads, we implement our OS modifications in Linux, define RMM hardware with respect to a recent Intel x86-64 Xeon core, and report overheads using a combination of hardware performance counters from application executions and functional TLB simulation.

RMM operating system prototype. We prototype the RMM operating system changes in Linux x86-64 with kernel v3.15.5. We implement the management of the range tables by intercepting all kernel memory-management operations. We implement range creation and eager paging by modifying the *mmap*, *brk* and *mremap* system calls. For our prototype range table, we implement a simple linked list rather than a B-tree. Because our applications spend only a tiny fraction of their time in the OS and the range TLB refill is not on the processor's critical path, this simplification does not affect our results.

We use a contiguity threshold of 32 KB (8 pages) to define the minimum size of a range translation. To increase the maximum size of a range, we increase the maximum allocation size in the buddy allocator to 2 GB, up from 4 MB by modifying the `max_order` parameter of the buddy allocator from 11 to 20. Because the default *glibc* memory management implementation does not coalesce allocations into fixed-size virtual ranges, we instead use the *TCMalloc* library [13]. In addition, we modify TCMalloc to increase the maximum allocation size from 256 KB to 32 MB.

RMM hardware emulation. We evaluate the RMM hardware described in Section 4.4 with Intel Sandy Bridge core shown in Table 4.4. We choose a 32-entry fully associative range TLB accessed in parallel with the L2 page TLB, since we estimate that it can meet the L2's

	Description
Processor	Dual-socket Intel Xeon E5-2430 (Sandy Bridge), 6 cores/socket, 2 threads/core, 2.2 GHz
Memory	96 GB DDR3 1066MHz
OS	Linux kernel version 3.15.5
L1 DTLB	4 KB pages: 64-entry, 4-way associative 2 MB pages: 32-entry, 4-way associative 1 GB pages: 4-entry, fully associative
L1 ITLB	4 KB pages: 128-entry, 4-way associative 2 MB pages: 8-entry, fully associative
L2 TLB	4 KB pages: 512-entry, 4-way associative 2 MB pages: —
range TLB	unrestricted sizes: 32-entry, fully associative

Table 4.4: System configurations and per-core TLB hierarchy.

timing constraints.

To measure the overheads of RMM, we combine performance counter measurements from native executions with TLB performance emulation using a modified version of BadgerTrap [51]. Compared to cycle-accurate simulation on these workloads, this approach reduces weeks of simulation time by orders of magnitude. Previous virtual memory system performance studies use the same approach [27, 28, 52].

BadgerTrap instruments x86-64 TLB misses. We add a functional range TLB simulator in the kernel that BadgerTrap invokes. On each page L2 TLB miss, BadgerTrap performs a range TLB lookup. Note that the actual implementation would perform the range TLB lookup in parallel, rather than after the L2 TLB miss. This emulation may thus underestimate the benefit of the range TLB, because the real hardware will install a missing page table entry, even if the virtual address hits in the range TLB. The actual RMM implementation reduces traffic to the L2 page TLB on range TLB hits, freeing up page TLB entries and potentially making it more effective. This simulation methodology may itself perturb TLB behavior. To minimize this problem, we allocate a 2 MB page in the kernel for the simulator itself, which reduces the differences with an unmodified kernel to less than 5%.

Note that in our evaluation of RMM in this chapter we use only 4 KB pages and range translations for the running applications, i.e., we disable the 2 MB (Transparent Huge Pages) and the 1 GB pages. The reason is that we focus here on the performance benefits

Performance Model

Ideal execution time	$T_{ideal} = T_{2M} - C_{2M}$
Average page walk cost	$AvgC_{4K/2M} = C_{4K/2M}/M_{4K/2M}$
Measured page walk overhead	$Over_{4K/2M} = C_{4K/2M}/T_{ideal}$
Simulated page walk overhead	$Over_{SIM} = M_{SIM} * AvgC_{4K}/T_{ideal}$
T: Total execution cycles	$M_{4K/2M}$: page walks with 4K/2M
C: Cycles spent in page walks	M_{SIM}: Simulated page walks

Table 4.5: Performance model based on hardware performance counters and BadgerTrap.

from range translations and range TLB, and thus want to avoid accounting for benefits from huge pages. However, there is no inherent limitation in combining range translations with huge pages in RMM (as we do in the evaluation of RMM in Chapter 5); simply some parts of the range translations are mapped with huge pages, instead of 4 KB pages, in the page table and the TLB support for huge pages is utilized as well.

Performance model. We estimate the impact of RMM on system performance with the following methodology. First, we run the applications on the real system (Table 4.4) with realistic input sets until completion and collect processor and TLB statistics using hardware performance counters. We use the Linux *perf* utility [11] to read the performance counters. We collect total execution cycles, misses for L2 TLB, and cycles spent in page walks. Based on these measurements we calculate (i) the ideal execution time (no virtual memory overhead), (ii) the measured overhead spent in page walks, and (iii) the estimated overhead with the simulated hardware mechanisms based on the fraction of reduced page walks, using a simple linear model [27, 52] given in Table 4.5.

Benchmarks. RMM is designed for a wide range of applications from desktop applications to big-memory workloads executing on scale-out servers. To evaluate the effectiveness of RMM, we select workloads with poor TLB performance from SPEC 2006 [59], BioBench [15], Parsec [32] and big-memory workloads [27] as summarized in Table 4.6. We execute each application sequentially on a single test machine without rebooting between experiments.

Suite	Description	Input	Memory
SPEC 2006	compute & memory intensive single-threaded workloads	astar	350 MB
		cactusADM	690 MB
		GemsFDTD	860 MB
		mcf	1.7 GB
		omnetpp	165 MB
		soplex	860 MB
PARSEC	RMS multi-threaded workloads	canneal	780 MB
		streamcluster	120 MB
BioBench	Bioinformatics single-threaded workloads	mummer	470 MB
		tigr	610 MB
Big memory	Generation, compression and search of graphs	Graph500	73 GB
	In-memory key-value cache	Memcached	75 GB
	NASA's high performance parallel benchmark suite.	NPB:CG	54 GB
	Random access benchmark	GUPS	67 GB

Table 4.6: Workload description and memory footprint.

4.8 Results

This section evaluates the cost of address translation, the impact of eager paging, and implications on energy of RMM, and shows substantial improvements in performance over current and proposed systems.

We compare RMM performance to the following systems. (i) We measure the virtual memory overheads of a commodity x86-64 processor (see Table 4.4) with 4 KB pages, 2 MB pages with transparent huge pages, and 1 GB pages with libhugetlbfs using hardware performance counters. (ii) We emulate multipage mappings in BadgerTrap, by implementing the Clustered TLB approach [97] of Pham et al., configured with 512 fully-associative entries. Each entry indexes up to an 8-page cluster, shown best by Clustered TLB [97]. We use eager paging to increase the opportunities to form multipages, improving on the original implementation. (iii) We emulate the performance of ideal direct segments. We assume all fixed-size memory regions that live for more than 80% of a program's execution time can be coalesced in a single contiguous range, which can be used to estimate the reduction in TLB misses with direct segment hardware [27].

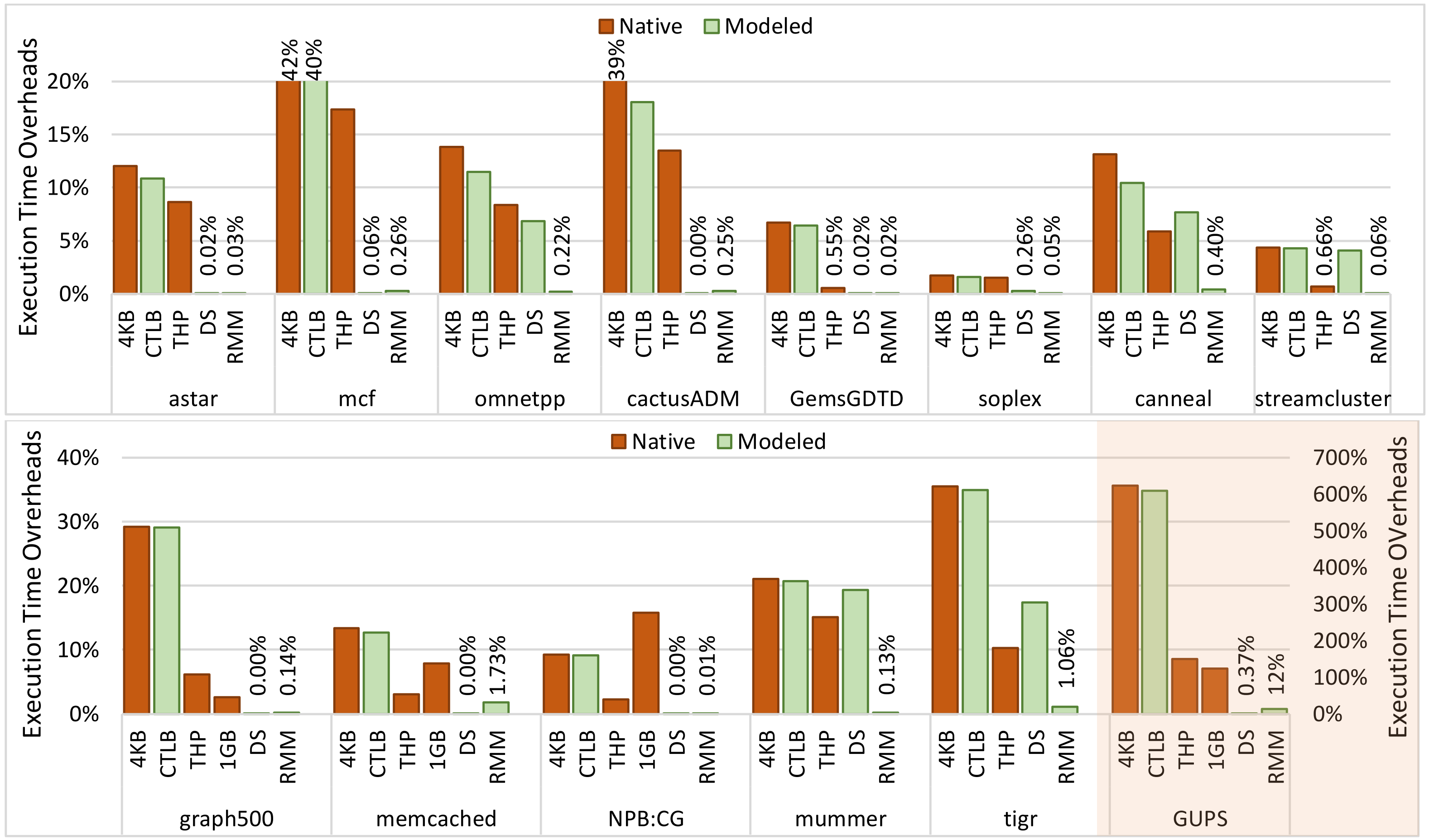


Figure 4.5: Execution time overheads due to page walks for SPEC 2006 and PARSEC (top) big-memory and BioBench (bottom) workloads. GUPS uses the right y-axis and thus shaded separately. 1GB pages are only applicable to big-memory workloads.

4.8.1 Performance analysis

Figure 4.5 shows the overhead spent in page walks for RMM compared to other techniques. The 4 KB, 2 MB Transparent Huge Pages (THP) [5] and 1 GB [8] configurations show the *measured* overhead for the three different page sizes available on x86-64 processors. All other configurations are emulated. The CTLB bars show Clustered TLB [97] results. The DS bars show direct segments [27] results and the RMM bars show the 32-entry range TLB results.

RMM performs well on all configurations for all workloads, improving substantially over all the other approaches, except direct segments. RMM eliminates the vast majority of page walks, significantly outperforms the Clustered TLB (CTLB), huge pages (THP and 1GB) and achieves similar or better performance than direct segments, but has none of its limitations. On average, RMM reduces the overhead of virtual memory to less than 1%.

For most workloads, the base page size (4 KB) incurs high overheads. For example, mcf, cactusADM, and graph500 spend 42%, 39% and 29% of execution time in page walks due to TLB misses. Even the applications with smaller working sets, such as astar, omnetpp, and mummer, still suffer substantial paging overheads using 4 KB pages.

Clustered TLB (CTLB) only offers limited reductions in overhead and only for small-memory workloads. CTLB performs better than 4 KB pages on small-memory workloads, such as cactusADM, canneal, and omnetpp. However, CTLB provides little benefit on big-memory workloads and performs worse than THP overall.

Huge pages (THP and 1 GB) reduce virtual memory overheads for all workloads but still leave room for improvement. The limited hardware support for huge pages (e.g., few TLB entries), poor application memory locality, and the mismatch of their sizes with the virtual memory contiguity all contribute to the remaining overheads. Note that recent processors provide more TLB entries for huge pages. However, this approach falls short: huge pages increase TLB reach by fixed size mappings and memory sizes increase more aggressively than TLB sizes. Hence, we believe that the virtual memory overheads that manifest in today's systems with 4 KB pages, will manifest similarly in tomorrow's systems with huge pages.

Direct segments achieve negligible overheads on big-memory workloads and some small-memory workloads. But, direct segments poorly serve workloads that require multiple ranges, such as omnetpp, canneal, or those that use memory-mapped files such as mummer.

In addition, direct segments requires application modifications and is not a transparent solution. Compared to direct segments, RMM is a better choice because it achieves similar or better performance on all workloads.

Redundant Memory Mappings achieve negligible overhead—essentially eliminating virtual memory overheads for many workloads. Only one workload has greater than 2% overhead, GUPS. As our sensitivity analysis in the next section shows, GUPS requires at least a 64-entry range TLB to achieve less than 1% overhead. Overall, RMM performs consistently better than the alternatives and in many cases eliminates the performance cost of address translation.

4.8.2 Range TLB sensitivity analysis

To achieve high performance, the range TLB must be large enough to satisfy most L1 TLB misses. Figure 4.6 shows the range TLB miss ratio as a function of the numbers of entries. We observe that a handful of workloads, such as cactusADM, memcached, tigr, and GUPS, suffer from high miss ratios with a 16-entry range TLB. Overall, a 32-entry range TLB eliminates more than 99% of misses for most workloads (97.9% on average), delivering a good trade-off of performance for the required area and power.

We also note that a single-entry range TLB is insufficient to eliminate virtual memory overheads. Most applications require multiple range table entries, especially those with large working sets, such as cactusADM, GemsFDTD and GUPS, and those with large numbers of ranges, such as memcached, mummer, and tigr. However, the single-entry results illustrate that the optional MRU Pointer would be effective at saving dynamic energy and latency in many cases. It reduces accesses to the range TLB by more than 50% for astar, omnetpp, canneal, streamcluster, and graph500.

4.8.3 Impact of eager paging

Eager paging increases range size by instantiating physical pages when the application allocates memory, rather than when the application first writes or reads a page. Table 4.7 shows the effect of eager paging on the number and size of ranges, and on time and memory overheads, compared to default demand paging. Default demand paging includes forming THPs, which we translate to ranges.

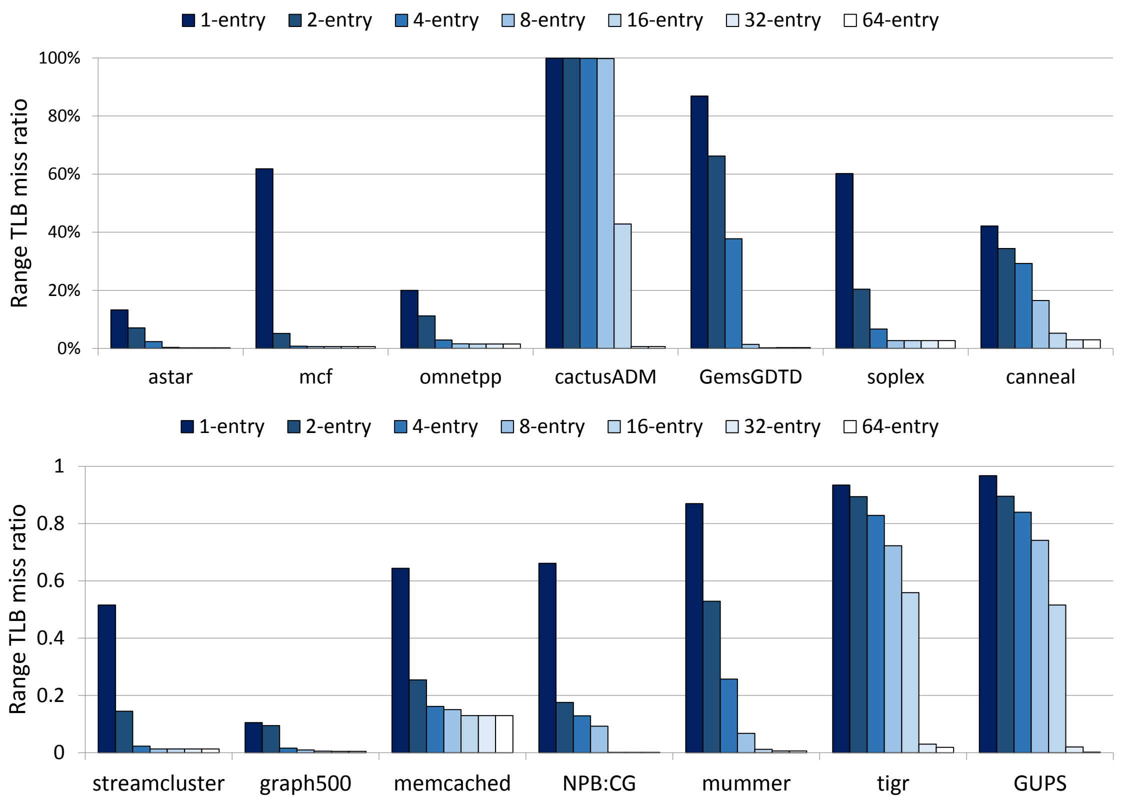


Figure 4.6: Sensitivity analysis of the range TLB miss ratio as a function of the number of range TLB entries.

The first two sections of Table 4.7 (demand paging and eager paging) compare the number of ranges, the percentage of the memory footprint covered by ranges with a contiguity threshold of 8 pages, and the range sizes (median, average, maximum) in terms of pages, created by demand and eager paging. Eager paging (i) lowers the median range size for small-memory workloads because it allocates fewer medium-sized ranges (the median for demand paging is usually 512, i.e., 2 MB regions, due to THP), (ii) increases the median range for big-memory workloads because it allocates fewer small and medium-sized ranges, and (iii) increases the average and maximum range size for all workloads because it allocates larger blocks from the buddy allocator. Overall eager paging generates orders of magnitude fewer ranges that cover a larger percentage of memory for all applications compared to demand paging. Thus eager paging assists in achieving high range TLB hit ratio with few entries.

	Demand Paging					Eager Paging					Demand vs. Eager	
			range size in 4 KB pages					range size in 4 KB pages				
Benchmark	# ranges	% memory	median	average	max	# ranges	% memory	median	average	max	% time overhead	% memory overhead
astar	170	94.52	512	478	1024	33	99.69	32	2810	8192	-1.15	8.14
mcf	449	99.72	512	957	4608	28	99.94	24	15637	262143	-4.10	1.58
omnetpp	91	96.30	512	438	512	27	99.03	20	1617	8192	-0.50	6.34
cactusADM	311	99.50	512	549	1024	70	99.84	8192	5537	8192	0.85	125.90
GemsFDTD	326	98.76	512	651	2048	61	99.75	256	3613	16384	11.65	2.74
soplex	333	98.32	512	633	4096	54	99.85	128	4502	81919	-1.78	13.45
canneal	410	95.96	202	453	1024	46	99.82	189	4248	32767	1.15	0.99
streamcluster	65	95.73	512	439	512	32	99.18	21	1122	16383	-1.61	21.41
mummer	837	85.51	32	120	512	61	99.68	512	1940	32768	-1.55	0.87
tigr	1149	95.16	16	123	1536	167	99.51	32	889	16384	-1.97	0.01
Graph500	18574	99.97	512	984	524288	32	99.99	2048	187236	524288	2.56	0.27
Memcached	1540	99.97	1024	29629	524288	86	99.99	2048	216857	524288	-3.95	0.17
NPB:CG	22746	99.98	512	586	1536	95	99.99	4096	146861	524288	0.87	4.56
GUPS	705	99.99	512	23823	524288	62	99.99	524288	271039	524288	-0.61	0.05

Table 4.7: Impact of eager paging on ranges, time, and memory compared to demand paging with Transparent Huge Pages.

Eager paging alters execution by changing when and how pages, even used pages, are allocated to physical memory. We measure execution overhead due to eager paging by running applications with the eager paging operating system support, but without the hardware emulation. Table 4.7 shows that the execution time for most applications is relatively unchanged. A few get faster: mcf and memcached improve by 4.1% and 3.9%. However, GemsFDTD degrades by 11%. In this case, the changes in physical page allocation affect cache indexing, increasing cache conflicts. Various orthogonal mechanisms address this problem [45, 106].

Eager paging anticipates that the application will use the requested memory regions and may thus increase the memory footprint. The last column of Table 4.7 reports the memory footprint increase with eager paging. Eager paging increases memory by a small amount for three of the big-memory workloads, and by less than 10% for 7 of the remaining 10 workloads. Eager paging increases memory substantially on cactusADM and NPB:CG (the percentage is low, but totals 2.3 GB), mainly because of instantiating memory that these applications request but never use, and because of modifying TCMalloc to increase contiguity. Thus RMM trades increased memory for better performance, a common tradeoff when memory is cheap and plentiful. Note that the OS can convert a range to pages or abandon ranges altogether under memory pressure as discussed in Section 4.6.

4.8.4 Energy

The primary RMM effect on energy is executing the application faster, which improves static energy of the system. According to our performance model, RMM improves performance by 2-84% and thus reduces the static energy by a similar ratio.

Secondary effects include the static and dynamic energy of the additional RMM hardware. The system accesses the range TLB in parallel with the L2 TLB, consuming dynamic energy on a L1 TLB miss. The dynamic energy of a 32-entry range TLB is relatively small with respect to the entire chip, and lower than of a fully-associative 128-entry L1 TLB (e.g., SPARC M7 [98]). Furthermore, replacing misses in the L2 TLB with hits in the range TLB saves dynamic energy by avoiding a page walk that performs up to four memory operations. The OS can identify workloads for which the range TLB provides little benefit and disable the range TLB (see Section 4.6), eliminating its dynamic energy.

To further explore power and energy impact of the range TLB on the address translation path, we implemented a 32-entry range TLB and a 512-entry L2 page TLB with search latency of six cycles in Bluespec. We then synthesized both designs with the Cadence RTL Compiler using 45nm technology (tsmc45gs standard cell library) at 3.49GHz under typical conditions. We specified that timing should be prioritized over area and power.* This analysis shows that the range TLB adds power that is less than half (39.6%) of L2 TLB's power. Moreover, the range TLB area is only 13% of the L2 TLB area. These results and the high range TLB hit ratio indicate that simply increasing the number of entries in the L2 TLB, which would also incur a cycle penalty on the critical path, at the same power and area budget will not be as effective as the RMM design.

4.9 Related Work

Virtual memory remains an active area of research. Previous work shows that limited TLB reach results in costly page walks that degrade application performance, often substantially [27, 29, 31, 52, 66, 77]. Section 4.2 described the qualitative differences between RMM and the most closely related work on multipage mappings (sub-blocked TLBs [111], CoLT [96], Clustered TLBs [97]), huge pages [5, 8, 94], and direct segments [27, 52], and Section 4.8 showed quantitatively that RMM substantially improves over them. Below we discuss other mechanisms that help reduce the overhead of TLB misses, and how they relate to RMM.

One common way to reduce the cost of a TLB miss is through accelerating the page walks. Commodity processors cache Page Table Entries (PTEs) in data caches to accelerate page walks [63]. Software-defined TLB structures, such as TSBs in SPARC [110] and software-managed sections of TLB in Intel Itanium [1], pin entries in the TLB to improve performance. MMU caches also reduce latency of page walks by caching intermediate levels of the page table, skipping one or more memory references during the page walk [23, 28, 61]. RMM is orthogonal to these approaches since it eliminates some page walks altogether. When page walks are required in RMM, these mechanisms can accelerate them.

Virtual memory overhead can also be reduced by lowering the number of TLB misses. For instance, the hardware can prefetch PTEs into the TLB in advance of their use [29, 76, 104]. However, the effectiveness of prefetching is limited by the predictability of the

*Due to license limitations, we synthesized memory cells of both structures with D flip-flops instead of SRAM cells.

memory access patterns. Alternatively, Barr *et al.* [24] proposed speculative translation based on huge pages. Similar to prefetching, this mechanism depends on the TLB behavior and favors sequential patterns. Last-level shared TLBs [31, 89] and cooperative TLBs [109] increase the TLB reach and reduce the number of page walks. Similarly, Papadopoulou *et al.* [95] proposed a prediction mechanism that allows all page sizes to share a single set-associative TLB. In addition, Du *et al.* [48] proposed mechanisms to allow huge pages to be formed even in the presence of retired physical pages. However, the total TLB reach is still limited for memory intensive applications since each TLB entry maps a single page unless ranges are used [77]. In contrast to these approaches, RMM generates and caches translations for arbitrarily large ranges. Thus RMM is less susceptible to irregularities in the application's access patterns and improves address translation for large memories.

Commercial processors have also used segmentation to implement virtual memory. The Burroughs B5000 [85] was an early adopter of pure segments. The 8086 [4] and iAPX 432 [60] processors also supported pure segmentation without paging. Later IA-32 processors provided segments on top of paging [66], but without any translation benefits for segments. In contrast to previous segmentation approaches, RMM combines the flexibility and robustness of paging while enjoying the translation performance of segmentation.

Prior work also proposes virtual caches to reduce the performance and energy overheads of the TLB by only translating after a cache miss [26, 66, 116]. However for those workloads that suffer many TLB misses due to poor locality, virtual caches just shift the translation to a lower level of the cache hierarchy while increasing the complexity of the system.

Finally, our proposed architecture resembles prior works in fine-grained memory protection [55, 113, 115], in the sense that both exploit range behavior. However, instead of exploiting only the contiguity of fine-grained protection rights across memory regions, RMM enhances and exploits the contiguity in memory allocation to accelerate address translation.

4.10 Summary

In this chapter we proposed Redundant Memory Mappings, a hardware/software co-designed implementation of virtual memory. RMM provides a novel and robust translation mechanism, that improves performance by increasing TLB reach and reducing the cost of virtual memory across all our workloads. RMM efficiently represents ranges of arbitrarily-many pages that are virtually and physically contiguous and layers this representation and its hardware redundantly to page tables and paging hardware. RMM requires only modest changes to existing hardware and operating systems. The resulting system delivers a virtual memory system that is high performance, flexible, and completely transparent to applications.

5 Energy-Efficient Address Translation

5.1 Introduction

Since their invention in the 1960s [44], TLBs have been a small monolithic structure and were able to deliver high performance. Commercial processors, however, keep on devoting more resources to memory and address translation to meet the ever increasing memory demands of memory intensive workloads. The common TLB organization found today includes multi-level TLBs with support for huge pages [14, 56, 107].

TLBs have been reported to consume a significant fraction of processor energy [2, 3, 49, 71, 72, 73]. The recent growth in the complexity of TLBs has further increased their energy consumption—a recent industrial report suggests that TLBs consume 3-13% of a processor's power [108].

The energy overheads associated with the TLBs come from two sources: (i) the *static energy of the chip* due to TLB misses that lead to longer execution times [51, 78], and (ii) the *dynamic energy* of TLB resources that are accessed to lookup the address translation on every memory operation. However, reducing the energy of address translation is not

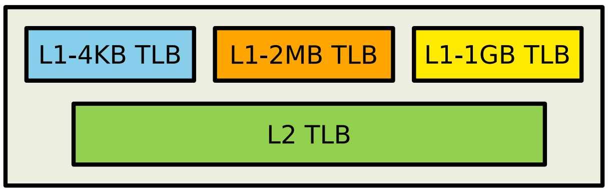


Figure 5.1: A common per-core two-level TLB organization that supports multiple page sizes (4 KB, 2 MB, and 1 GB) through separate L1 TLBs. All L1 TLBs are accessed on every memory operation, increasing the dynamic energy spent in address translation.

straightforward. When the static energy of the chip decreases due to fewer TLB misses, the dynamic energy of the TLBs increases due to the augmented complexity that ensures the low TLB miss ratio.

Prior research focused on reducing the dynamic energy of TLBs through various techniques, such as optimizing TLB circuits [71], partitioning TLBs [21, 38, 41, 82], filtering accesses to TLBs [22, 38, 49], dynamically resizing monolithic TLBs [20], virtual caches to access TLBs on L1 cache misses [26, 66, 116], and selectively avoiding TLB accesses [72, 73, 74]. However, these energy optimization techniques do not take into account hardware support for increasing the TLB reach (e.g., huge pages and range translations), that primarily targets improving performance and reducing static energy overheads due to TLB misses. Only the recent work on TLB$_{Pred}$ [95] considers huge pages for improving the dynamic energy efficiency in TLBs. The performance of TLB$_{Pred}$ depends on huge pages successfully reducing misses, but prior work shows that huge pages can still incur high performance overheads due to TLB misses [27, 28, 77]. In response, researchers proposed techniques that further increase the TLB reach [27, 51, 78, 96, 97, 111] to overcome the limitations of huge pages.

The goal of this work is to improve the energy efficiency of address translation in the presence of mechanisms that increase TLB reach.

Towards that goal, we perform energy characterization of the address translation path. We use a common TLB organization, found in Intel x86-64 processors as our baseline, that includes a per-core two-level TLB hierarchy, with a separate set associative L1 TLB for each supported page size, e.g., for 4 KB, 2 MB, and 1 GB pages, as shown in Figure 5.1. Our analysis shows that the L1 TLBs are the primary source of dynamic energy spent in address translation. We also find that page walks consume significant amount of energy with 4 KB pages. While huge pages and other techniques that increase TLB reach [27, 51, 78, 96, 97, 111] reduce the energy due to page walks, we observe that the "innocent" L1 TLB remains the dominant source of dynamic address translation energy, because separate L1 TLBs are accessed on every memory operation.

Our approach for providing energy-efficient address translation is driven by the following key observation: simply accessing all L1 TLB resources might not improve performance, because not all L1 TLBs contribute the same to hits, especially when techniques that increase the TLB reach are employed.

We propose *Lite*, an opportunistic mechanism that targets commodity processors with TLB support for huge pages. Lite monitors the utility of ways in the L1 TLBs for each page size based on the distance of TLB hits from the least-recently-used (LRU) position in an interval fashion, similar to the accounting cache [47] and utility-based cache partitioning [102]. At the end of each interval, Lite evaluates the utility of L1 TLBs. In case the utility of some active ways is insignificant, Lite downsizes each L1 TLB individually by disabling ways [16]. Lite thus accesses fewer ways in the L1 TLBs, saving energy at the cost of introducing a few additional misses. The resulting TLB$_{Lite}$ organization requires minimal modifications and opportunistically reduces L1 TLB energy with negligible impact on performance.

We additionally propose RMM$_{Lite}$ to further augment the potential of Lite for reducing the energy in L1 TLBs while at the same time reducing both the energy and performance overheads due to L1 TLB misses. RMM$_{Lite}$ builds on Redundant Memory Mappings (RMM) that was presented in Chapter 4. and considered only L2-range TLBs for increasing the TLB reach and reducing the number of page walks. In this chapter, we introduce to RMM an L1-range TLB and add the Lite resizing mechanism to the L1-page TLBs. The L1-range TLB is accessed in parallel with the L1-page TLBs and is small (e.g., 4 entries) in order to meet the tight timing requirements of L1 TLBs, yet the L1-range TLB is powerful. Each range TLB entry can hold a mapping of unlimited size, that enables the L1-range TLB to enjoy a high hit ratio. Therefore, Lite downsizes L1-page TLBs more aggressively without affecting performance. Overall, RMM$_{Lite}$ improves both energy efficiency and performance of address translation.

To evaluate the proposed TLB$_{Lite}$ and RMM$_{Lite}$ designs, we developed a TLB simulator based on Pin [88], `pagemap` [10], and Cacti [83] and we ran various TLB intensive workloads from Spec2006 [59], BioBench [15], and Parsec [33]. Our findings show that TLB$_{Lite}$ reduces the dynamic energy spent in address translation by 23% while slightly increasing the cycles spent in TLB misses (from 16.6% to 17.2%) compared to huge pages [5]. RMM$_{Lite}$ reduces the dynamic energy spent in address translation by 71% on average compared to huge pages. Above the near-zero L2 TLB misses from RMM, RMM$_{Lite}$ further reduces the overhead from L1 TLB misses by 99%.

In summary, the main contributions of this chapter are:

- We characterize the dynamic energy spent in address translation, and identify the L1 TLBs and page walks as the most significant sources.
- We show that simply accessing all L1 TLBs might not improve performance in the presence of huge pages.
- We propose Lite to reduce the energy spent in L1 TLBs by opportunistically disabling resources with low impact on performance, and apply it to a standard TLB hierarchy with support for huge pages (TLB$_{Lite}$).
- We propose RMM$_{Lite}$, that adds to RMM an L1-range TLB and Lite, to reduce further the energy and performance overheads spent in L1 TLBs leveraging the efficient representation of range translations.

The organization of the rest of the chapter is as follows: Section 5.2 provides background information on address translation, Section 5.3 analyzes the sources of dynamic energy spent in address translation, Section 5.4 quantifies our key observation and presents the proposed designs, Section 5.5 describes our evaluation methodology, Section 5.6 provides the results, Section 5.7 discusses related work, and Section 5.8 summarizes this chapter.

5.2 Background

This section highlights some trends for improving TLB performance, and then outlines some characteristics that are found in commodity processors. Note that although we focus on the x86-64 architecture, the proposed solutions apply to other architectures that include TLB support for huge pages.

5.2.1 Trends in TLBs

Two-level TLBs form a common organization for address translation in today's processors [56, 107]. The L1 TLB is usually small (e.g., 64 entries) and features a very fast search operation (1-2 cycles), while the L2 TLB is usually larger (e.g., 512 entries) and holds more translations at the cost of increased access latency (∼7 cycles [63]).

Huge Pages [5, 8] increase the TLB reach and reduce the performance overhead of page walks [27, 28, 77, 95] by mapping a large fixed size region of memory with a single TLB entry [61, 91, 101, 107]. The hardware support for huge pages usually includes either a separate set associative L1 TLB for each page size, as in Intel processors [56], or a single fully associative L1 TLB that supports both 4 KB and huge pages, as in SPARC and AMD processors [14, 107]. These two approaches dominate because supporting all page sizes in a single set associative TLB is not straight-forward: the page size defines the index bits to access the TLB, but the page size is unknown during the TLB lookup time [95, 112]. Separate set associative TLBs are generally more energy-efficient as compared to fully associative TLB. As we focus on energy, our baseline in this work assumes the more efficient separate set associative L1 TLBs.

5.2.2 Summary

Table 5.1 overviews the details of the per-core TLB hierarchy for the recent Sandy Bridge and Haswell, and the forthcoming Broadwell x86-64 processors. We observe that all these processors have a two-level TLB organization, with support for various pages sizes by separate individual L1 TLBs. This data suggests their recipe for improving TLB performance consists of having separate L1 TLBs for various page sizes, and increasing the size of the L2 TLB which is off the critical path.

To summarize, the TLB resources become larger and more complex to meet the increasing TLB demands of memory intensive workloads. However the performance and static energy improvements come at the cost of accessing multiple structures and increasing dynamic energy. Our approach reduces the dynamic energy spent in address translation by leveraging mechanisms that increase TLB reach, such as huge pages and range translations, making the case for energy-efficient address translation.

L1 DTLBs			**L2 DTLBs**								
Sandy Bridge / Haswell / Broadwell			**Sandy Bridge**			**Haswell**			**Broadwell**		
Page-size	Entries	Assoc.	Page-size	Entries	Assoc.	Page-size	Entries	Assoc.	Page-size	Entries	Assoc.
4 KB	64	4-way	4 KB	512	4-way	4 KB/2 MB	1024	8-way	4 KB/2 MB	1536	12-way
2 MB	32	4-way	2 MB	—							
1 GB	4	fully	1 GB	—		1 GB	—		1 GB	16	4-way

Table 5.1: Details of the private, per-core, data TLB hierarchy for the three latest Intel processor architectures.

5.3 Energy Characterization

In this section we analyze the sources of dynamic energy spent in address translation. We first provide an overview of our methodology, and then we analyze where the dynamic energy is spent with 4 KB pages, huge pages, and RMM.

5.3.1 Methodology Overview

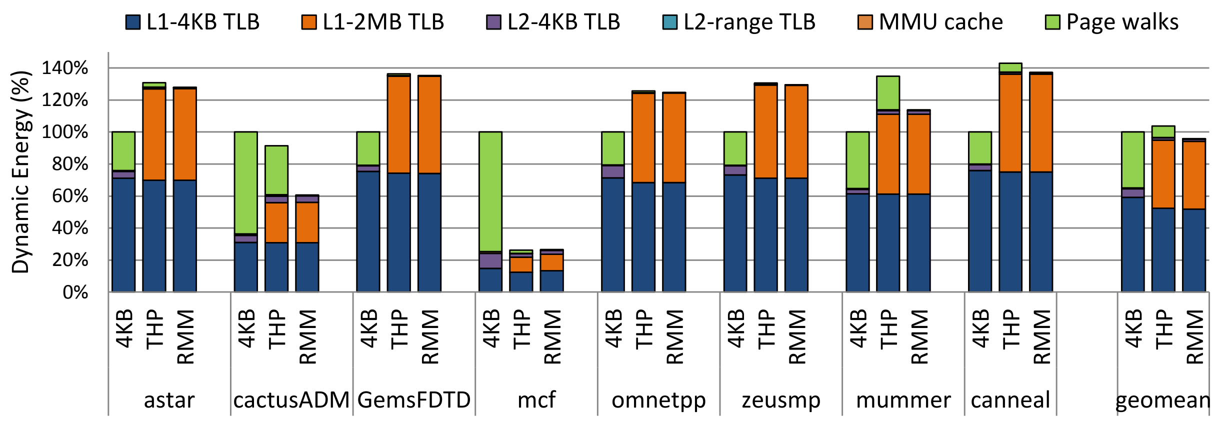
For the purposes of this study, we developed a detailed TLB simulator based on Pin [88], `pagemap` [10], and Cacti [83]. We model a private, per-core, two-level TLB organization backed by an MMU cache. The configuration and the parameters are based on those of an x86-64 Intel Sandy Bridge processor and summarized in Table 5.1.

We assume the existence of a mechanism that statically disables accesses to TLB resources that are not used. For instance, the L1-2MB TLB and L1-1GB TLB could be disabled for a running process that uses only 4 KB pages and no 2 MB or 1 GB pages. Such a mechanism could be easily implemented in hardware; a mask would enable lookups in the L1-2MB TLB only after a 2 MB page table entry has been fetched by a page walk. In this study we assume the existence of such mechanism, and thus unused TLB structures do not account for the dynamic energy overhead.

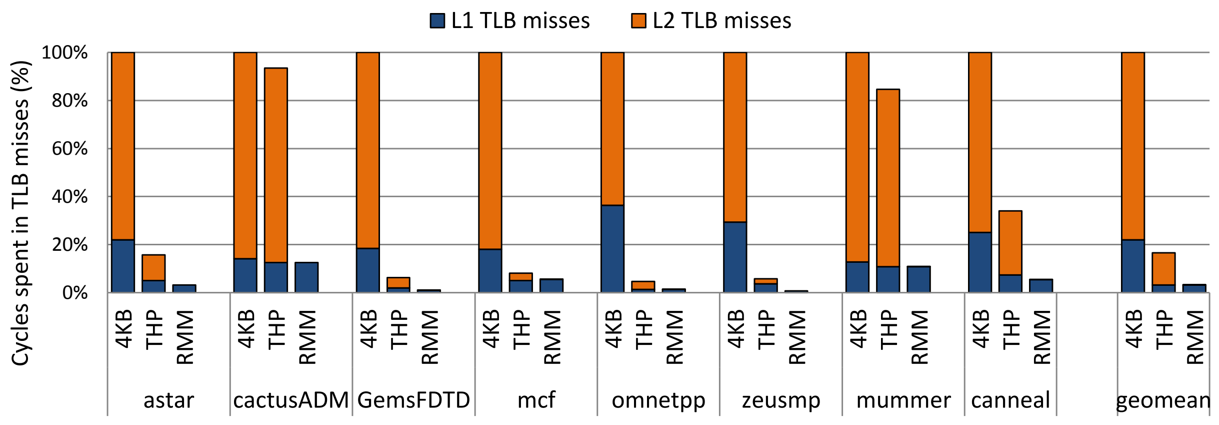
We fast-forward execution for 50 billion instructions and then simulate for the next 50 billion instructions. More details about our methodology can be found in Section 5.5.

5.3.2 Where is the energy spent?

Figures 5.2a and 5.2b break down the dynamic energy spent in address translation and the cycles spent in L1 and L2 TLB misses for various workloads with the following three configurations: (i) *4KB* supports 4 KB pages, (ii) *THP* supports both 4 KB and 2 MB pages with transparent huge pages [5], and (iii) *RMM* supports 4 KB, 2 MB pages, and range translations with a 32-entry fully associative L2-range TLB [78]. We assume optimistically that all page walk references hit always in the L1 cache of the memory hierarchy with respect to the dynamic energy, because additional cache misses in the memory hierarchy can greatly hurt energy due to accessing more and larger caches. The results are normalized to the dynamic energy spent with 4KB pages per workload.



(a) Dynamic energy spent in address translation (%)



(b) Cycles spent in L1 and L2 TLB misses (%)

Figure 5.2: Dynamic energy spent in address translation (a) and cycles spent in TLB misses (b) for the execution of 50 billion instructions with three configurations: (i) *4KB* supports only 4 KB pages, (ii) *THP* supports both 4 KB and 2 MB pages with transparent huge pages [5], and (iii) *RMM* supports 4 KB, 2 MB pages, and range translations with an L2-range TLB [78]. The results are normalized to those with 4 KB pages per workload. The two major sources of dynamic energy overhead with 4 KB pages are the L1 TLBs and the page walks. THP and RMM reduce the energy and cycles spent in page walks, but increase the total dynamic energy spent in address translation because multiple L1 TLBs are accessed on every memory operation.

We identify two major sources of dynamic energy overhead with 4KB and THP configurations:

1. **L1 TLBs energy consumption**. To make address translation as fast as possible, the processor accesses *all L1 TLB structures*, i.e., the L1-4KB TLB, the L1-2MB TLB, and the L1-1GB TLB, in parallel on *every memory operation*. Consequently, the L1 TLBs consume 60% and 91% of dynamic energy with 4KB and THP. We further identify the L1-4KB TLB as the most dominant source of dynamic energy (50% of dynamic energy with THP) due to its larger size compared to the other L1-page TLBs.

2. **Page walk energy consumption**. On a TLB miss at every TLB level, the page table hardware walks the page table, which requires multiple memory accesses (e.g., 4, 3, and 2 memory accesses for 4 KB, 2 MB, and 1 GB pages) that incur performance and energy penalties. This source of energy overhead becomes more prevalent (i) for applications that suffer frequently from page walks, such as cactusADM and mcf, and (ii) as the page walk references hit less in the L1 cache. Figure 5.3 quantifies the impact of page walk locality in the dynamic energy as the L1 cache hit ratio for the page walk references reduces from 100% (all references hit in L1 cache) to 0% (all references miss in L1 cache, but hit in L2 cache). The dynamic energy may increase by up to 91% for mcf, due to poor page walk locality in the cache hierarchy.

5.3.3 Do huge pages help?

We observe that THP reduces the cycles spent in TLB misses by 83% on average compared to 4KB. However, THP affects the dynamic energy of address translation in a less straightforward way compared to performance. With THP, the dynamic energy of address translation decreases only for cactusADM and mcf, and increases for all other workloads. This happens because THP reduces the number of page walks and their portion in dynamic energy along with static energy by completing the workload faster, as explained next in Section 5.3.6. However, this saving occurs at the cost of accessing one extra L1 TLB for 2MB pages on every memory operation, which in turn increases the dynamic energy spent for address translation in the L1-page TLBs. Overall, THP increases the dynamic energy spent in address translation by up to 43% for canneal and by 4% on average, compared to 4 KB pages.

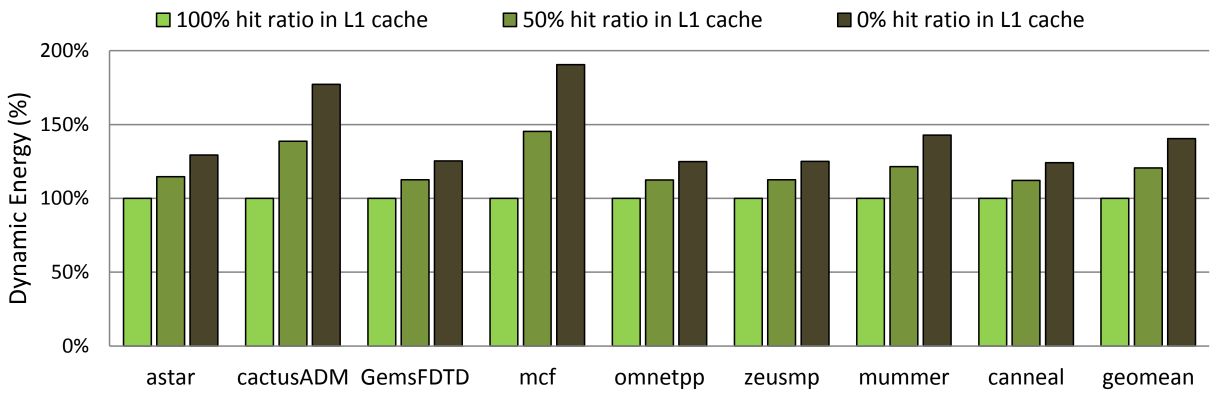


Figure 5.3: Sensitivity analysis of the dynamic energy spent in address translation, ranging the L1 cache hit ratio from 100% (all accesses hit in L1 cache) to 0% (all accesses miss in L1 cache but hit in L2 cache) for the page walk references with 4 KB pages. The locality of page walks significantly affects the dynamic energy.

5.3.4 Does RMM help?

The RMM configuration has the same TLB organization as THP, including a 32-entry L2-range TLB. In addition, the RMM configuration assumes perfect eager paging, i.e., the operating system perfectly allocates all contiguous pages of virtual address space to contiguous physical pages. We observe that RMM eliminates almost completely the page walks, and reduces by 96% the cycles spent in TLB misses compared to 4KB. However, RMM incurs high dynamic energy overhead (only 4% less on average compared to 4KB), as the access pattern to the L1 TLBs is similar to THP.

5.3.5 Do larger TLB organizations help?

Our energy characterization here uses as baseline the TLB organization of an x86-64 Intel Sandy Bridge processor. However, as we explained in Section 5.2, newer processors increase TLB reach—by providing more L2 TLB entries that may hold either 4 KB or 2 MB pages—but the L1 TLBs have remained stagnant (Table 5.1). Thus, the energy and performance results for such larger TLB organizations would look similar to those for huge pages: (i) lower dynamic energy and performance overheads due to page walks (because of the increased L2 TLB reach), but (ii) still high dynamic energy overhead due to the lookups in the separate L1 TLBs.

5.3.6 Discussion

The total energy consumption is the sum of static and dynamic energy. Since huge pages and range translations (and other techniques that increase TLB reach [27, 51, 96, 97, 111]) enable most applications to execute faster, they also decrease the static energy of the system. However, optimizing for energy efficiency requires addressing both dynamic and static sources of energy. Thus, in addition to reducing the execution cycles and the static energy, in this thesis we focus on reducing the dynamic energy spent in address translation.

5.4 Efficient Address Translation

An ideal solution for energy-efficient address translation would reduce the energy spent in L1 TLB accesses and page walks with negligible impact on performance. To provide energy-efficient address translation, we propose:

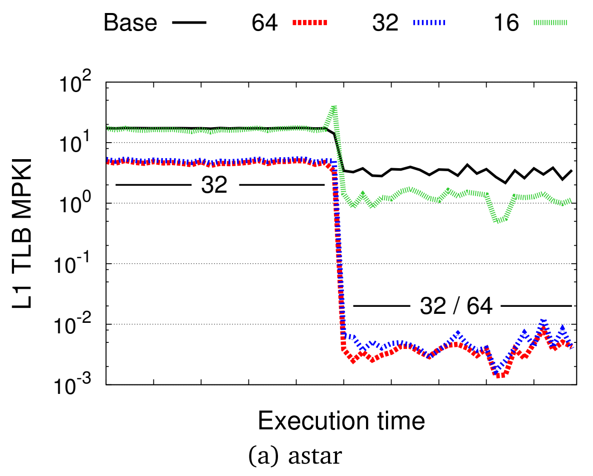
- **Lite**, a mechanism that monitors the utility of ways in all L1-page TLBs and adaptively changes their size with way-disabling [16]. The resulting **TLB**$_{Lite}$ organization opportunistically reduces L1 TLB energy with negligible impact on performance, and requires minimal modifications to commodity processors.

- **RMM**$_{Lite}$, a novel TLB organization that leverages the powerful representation of range translations in RMM [78]. RMM$_{Lite}$ adds an L1-range TLB and the Lite mechanism to RMM. The high hit ratio in the L1-range TLB allows Lite to further reduce the energy spent in L1-page TLBs and reduce significantly the total energy and performance overheads of L1 TLB misses.

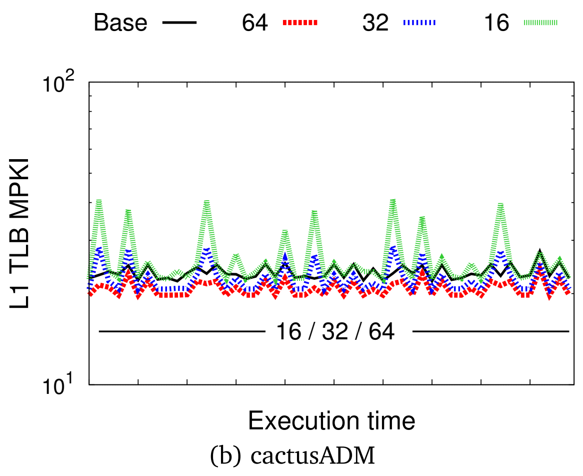
5.4.1 Opportunity

Our approach is based on the question: *"Do we need to access all L1 TLB resources on every memory operation?"* For example, if the hits in L1 TLBs are dominated by those entries for 2 MB pages, then the L1-4KB TLB could be dynamically downsized to reduce the dynamic energy spent in L1 TLBs without affecting performance, and vice versa.

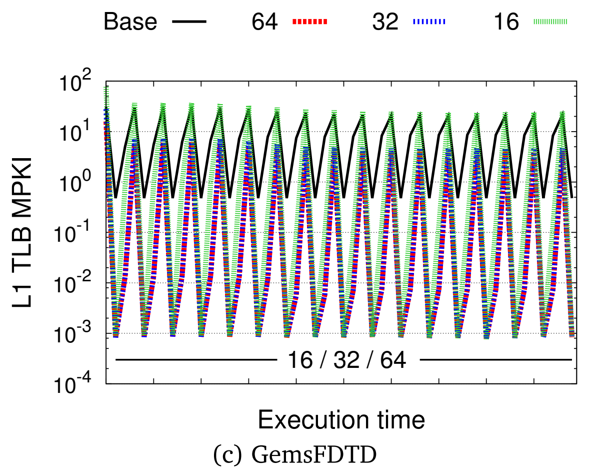
To quantify our hypothesis, we profile the performance of L1 TLBs with transparent huge pages enabled [5], when a smaller L1-4KB TLB with fixed size is employed during



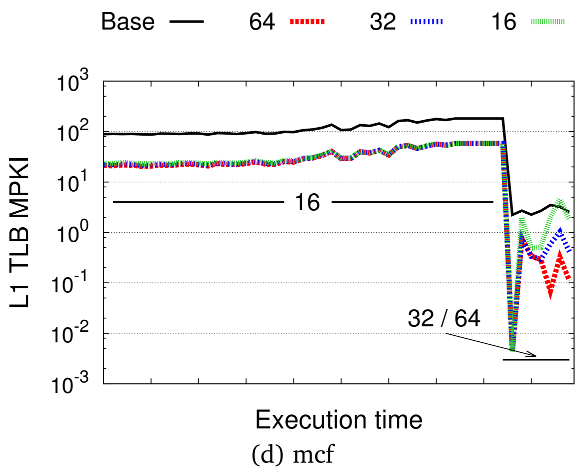
(a) astar



(b) cactusADM

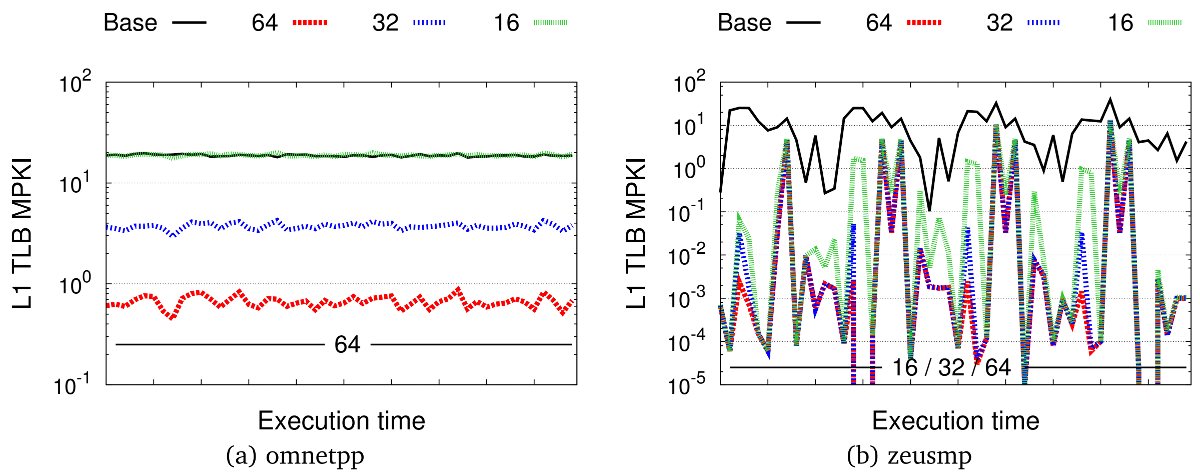


(c) GemsFDTD

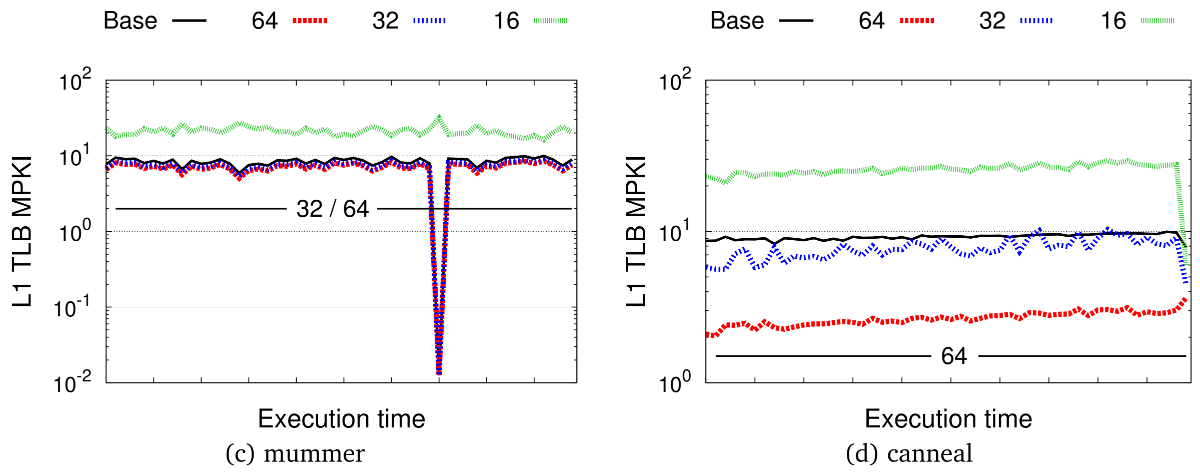


(d) mcf

Figure 5.4: L1 TLB misses per thousand instructions (MPKI) (aggregated for all L1 TLBs) during the execution of 50 billion instructions for astar, cactusADM, GemsFDTD, and mcf, with the following four configurations: (i) *Base* supports only 4 KB pages (same as *4KB* in Section 5.3), (ii) *64* supports both 4KB and 2 MB pages (same as *THP* in Section 5.3), (iii) *32* has the same configuration as *64* but with 32-entry 2-way L1-4KB TLB, and (iv) *16* has the same configuration as *64* but with 16-entry direct-mapped L1-4KB TLB. We observe that most workloads exhibit similar performance even with smaller L1-4KB TLBs in the presence of huge pages, but there is no single TLB configuration that is optimal for all workloads and during all execution time.



(a) omnetpp (b) zeusmp



(c) mummer (d) canneal

Figure 5.5: L1 TLB misses per thousand instructions (MPKI) (aggregated for all L1 TLBs) during the execution of 50 billion instructions for omnetpp, zeusmp, mummer, and canneal, with the following four configurations: (i) *Base* supports only 4 KB pages (same as *4KB* in Section 5.3), (ii) *64* supports both 4KB and 2 MB pages (same as *THP* in Section 5.3), (iii) *32* has the same configuration as *64* but with 32-entry 2-way L1-4KB TLB, and (iv) *16* has the same configuration as *64* but with 16-entry direct-mapped L1-4KB TLB. We observe that most workloads exhibit similar performance even with smaller L1-4KB TLBs in the presence of huge pages, but there is no single TLB configuration that is optimal for all workloads and during all execution time.

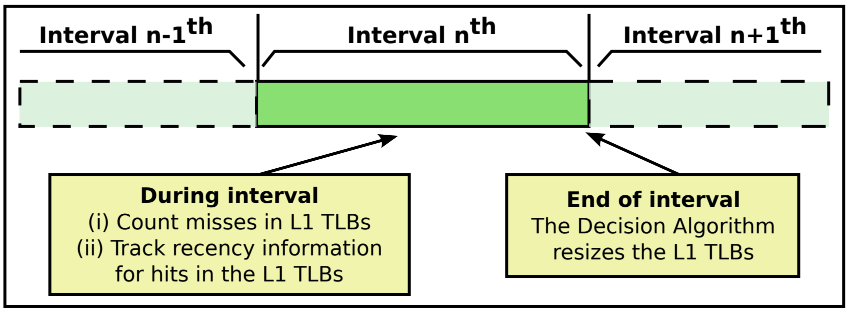

Interval n-1th
Interval nth
Interval n+1th
During interval
(i) Count misses in L1 TLBs
(ii) Track recency information for hits in the L1 TLBs
End of interval
The Decision Algorithm resizes the L1 TLBs

Figure 5.6: Lite divides the execution time of an application into *intervals*. During each interval, Lite tracks the performance of L1 TLBs. At the end of each interval, Lite decides whether to resize the L1 TLBs.

the execution. We assume that the L1-4KB TLB becomes smaller by reducing ways in powers-of-two while the number of sets remains constant. Figures 5.4 and 5.5 show the misses in the L1 TLBs per thousand instructions (MPKI) during the execution of 50 billion instructions. Configurations *64*, *32*, and *16* employ a 64-entry 4-way, a 32-entry 2-way, and a 16-entry direct-mapped L1-4KB TLB, respectively. The L1-2MB TLB is 32-entry 4-way for all configurations.

We find that most workloads exhibit similar performance even with smaller L1-4KB TLBs in the presence of huge pages. However, there is no single TLB configuration that is optimal for all workloads. For example, astar and mcf require configuration *16*, while cactusADM, GemsFDTD, and mummer require configuration *32*, to provide similar performance as with configuration *64* that runs with all L1 TLB resources enabled. In addition, a single TLB configuration is often not the optimal during the workload's total execution due to phased behavior. For example, astar, GemsFDTD, and mcf require different configurations to preserve similar performance. Thus, a mechanism that dynamically resizes the L1 TLBs is required to adapt to the workload.

5.4.2 The *Lite* Mechanism

Lite dynamically adapts the size of L1 TLBs to reduce their dynamic energy. Lite consists of three components: (i) the *monitoring mechanism* that tracks the actual performance of L1 TLBs and estimates the utility of all L1 TLBs, (ii) the *decision algorithm* that decides whether to change the size of the L1 TLBs, and (iii) the *reconfiguration mechanism* that configures the size of L1 TLBs.

Monitoring TLBs

Lite tracks the performance of the L1 TLBs in the *actual-misses-counter* for an interval. The counter is increased on every translation lookup that misses in L1 TLBs of that core and that triggers an access to the L2 TLB.

Lite estimates the cost of way-disabling by tracking the utility of all active ways for each L1 TLB in powers-of-two. Lite leverages the LRU replacement policy and relies on the distance of TLB hits from the LRU position in each set to estimate the utility of ways, similar to the accounting cache [47] and utility-based cache partitioning [102]. Lite introduces *lru-distance-counters* per L1 TLB. Since Lite disables ways in powers-of-two, we only need $[log_2(n)+1]$ lru-distance-counters for each n-way set-associative L1 TLB. Figure 5.7 shows Lite for an 8-way L1 TLB. The corresponding lru-distance-counter is increased on every L1 TLB hit: a hit with distance 7, 6, 4-5, or 0-3 from the LRU position increases the lru-distance-counters [0], [1], [2], or [3], correspondingly. In this way, each lru-distance-counter holds the number of TLB misses that would have occurred, if those ways were disabled. Note that when less ways are active, the corresponding lru-distance-counters are not used, because there are no tags to keep track of activity.

Finally, Lite keeps the actual number of L1 TLB misses of the previous interval in the *previous-misses-counter* to respond to TLB performance degradation, as explained next.

The Decision algorithm

Algorithm 5.1 shows the simplified pseudocode for the decision algorithm of Lite. Lite resizes all L1-page TLBs (4KB, 2MB, and 1GB) of each core's TLB organization separately. The algorithm uses the number of L1 TLB misses per thousand instructions (MPKI) to estimate the performance of the L1 TLBs and the utility of the active ways in each L1 TLB.

Disabling ways. At the end of each interval, Lite estimates for each L1-page TLB (4KB, 2MB, and 1GB) the potential MPKI if way disabling had been applied to the currently active ways. To achieve this, Lite uses the actual-misses-counter and the lru-distance-counters. In case the potential MPKI for fewer ways does not significantly increase compared to the actual MPKI, based on a threshold ϵ, then Lite disables those ways for that L1 TLB. During next interval, the resized L1 TLBs will save dynamic energy on every memory access.

Activating ways. Lite profiles only the active ways and therefore can reason only for them.

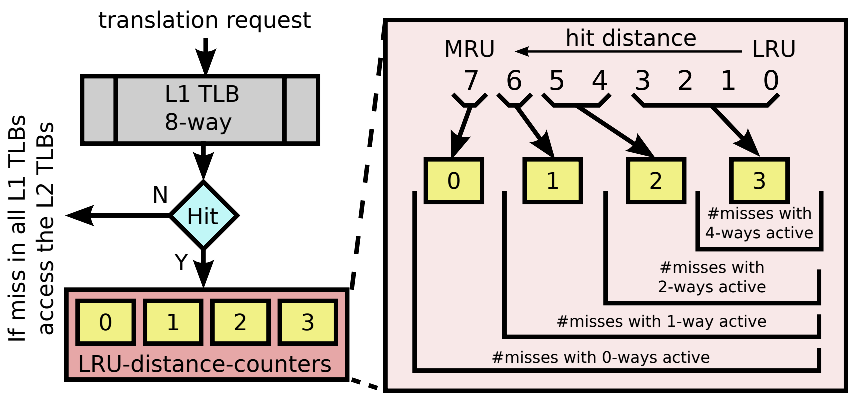

translation request
L1 TLB
8-way
Hit
N
Y
If miss in all L1 TLBs access the L2 TLBs
0
1
2
3
LRU-distance-counters
MRU
hit distance
LRU
7 6 5 4 3 2 1 0
#misses with 4-ways active
#misses with 2-ways active
#misses with 1-way active
#misses with 0-ways active

Figure 5.7: Lite introduces *lru-distance-counters* per L1 TLB to track the utility of ways [47, 102]. The corresponding counter is increased on each L1 TLB hit depending on the distance from the LRU position. At the interval end, each counter holds the number of L1 TLB misses that would have occurred, if those ways were disabled.

However, Lite is unaware whether more than the active ways would be really useful. For example, consider an 8-way L1 TLB that currently runs with 4 active ways; Lite sees that the potential MPKI does not significantly change whether using 4 or 2 active ways, and decides to use 2 ways. However, if all 8 ways were active, the potential MPKI could be significantly lower and Lite would have not decided to apply way-disabling. The problem becomes even more prevalent when the 1-way configuration is used, as no alternatives are evaluated. To respond in such cases, Lite randomly activates all the ways in all L1 TLBs based on a probability. The randomly introduced variability allows also Lite to avoid pathological cases in which the decisions synchronize with non-representative phases of the application, that may lead to poor decisions.

Finally, Lite activates all ways in the L1 TLBs when their performance degrades, e.g., when the application experiences phased TLB behavior, or the operating system breaks huge pages to 4 KB pages to respond to memory pressure. Lite records the actual MPKI of the previous interval, and compares it to the actual MPKI of the current interval. In case the MPKI surpasses the defined threshold ϵ, Lite activates all ways in the L1 TLBs.

Threshold. The threshold ϵ can either be a relative percentage increase or an absolute value increase of MPKI with respect to the reference value, i.e., the MPKI with all ways activated in L1 TLBs. The choice depends on the reference value itself. A relative percentage is preferable for high reference values (e.g., more than 1 MPKI) to control Lite's impact.

ALGORITHM 5.1: Pseudocode of Lite's decision algorithm.

```
// at the end of each interval;
compute the actual_mpki based on actual_misses_counter;
if (random probability is triggered) then
    activate all ways in L1 TLBs;
else if (actual_mpki - previous_mpki $\geq \epsilon$) then
    activate all ways in L1 TLBs;
else
    potential_misses = actual_misses;
    for each L1 TLB of that core do
        for (i = $log_2$(active_ways); i $\geq$ 1; i–) do
            potential_misses += lru_distance_counter[i];
            compute the potential_mpki based on potential_misses;
            if (potential_mpki - actual_mpki < $\epsilon$) then
                disable half of the active ways;
            else
                potential_misses -= lru_distance_counter[i];
                break;
            end
        end
    end
end
if (previous_mpki > actual_mpki) then
    previous_mpki = actual_mpki;
end
```

An absolute value is preferable for lower reference values, because even though the MPKI increases with respect to the reference value, the MPKI remains still negligible and, thus, Lite correctly decides to disable ways.

Reconfiguring TLBs

Lite reconfigures the L1 TLBs through way-disabling [16]. Way-disabling requires that the memory structure be partitioned into subarrays. We assume that such partitions are either already present in L1-TLBs for both timing and energy reasons, or can be easily implemented with minor circuit modifications. With way-disabling, only the active ways are searched in each TLB lookup, and thus the dynamic energy spent in address translation reduces.

Consistency. TLBs are read-only structures and do not hold dirty data. Thus, when Lite deactivates ways in a TLB, no write-back operations are necessary. Lite only invalidates translations in the disabled ways, so that when these ways are re-activated, they will not hold any stale translations.

5.4.3 RMM_{Lite} for Energy-Efficient TLBs

We also propose to add Lite and an L1-range TLB to Redundant Memory Mappings (RMM) [78] to further reduce the energy and performance overheads of L1 TLBs. As described in Chapter 4, RMM uses *range translations*, an efficient, alternative representation of arbitrarily large ranges of pages that are contiguously allocated in both virtual and physical address space. RMM targets reducing the number of page walks and employs an L2-range TLB that is accessed in parallel with the L2-page TLB.

RMM_{Lite} augments the RMM with a small *L1-range TLB* and adds the Lite mechanism to the L1-page TLBs. The L1-range TLB is accessed on every memory operation in parallel with the L1-page TLBs. The L1-range TLB is fully associative and very small, e.g., 4 entries like the small L1-1GB TLB, so that it meets the tight timing requirements of L1 TLBs. The functionality and organization of L1-range TLB is the same as the originally proposed L2-range TLB, i.e., it caches a small number of range translations. Figure 5.8 shows the proposed TLB hierarchy for energy-efficient address translation. On an L1-range TLB hit, the address translation is obtained fast. On an L2-range TLB hit (after a miss in L1 TLBs), the hit range translation entry is copied to the L1-range TLB, in addition to copying the corresponding page table entry in the L1-page TLBs as in RMM. Note that the L1 TLBs for huge pages could either be simply disabled by the naive mechanism that was discussed in Section 5.3.1, or completely replaced by the (possibly larger) L1-range TLB.

The L1-range TLB itself increases the dynamic energy spent in L1 TLB accesses, because one more TLB is accessed on every memory operation. In addition, the L1-range TLB performs range checks instead of equality checks as in translation for fixed size memory regions. Thus, an L1-range TLB access costs more than an L1-page TLB access in terms of energy. However, each L1-range TLB entry maps an arbitrarily large range of contiguously allocated pages. The L1-range TLB can achieve higher hit ratio compared to page TLBs. This increased hit ratio in the L1-range TLB reduces further the utility of the L1-page TLBs. In response, Lite disables ways more aggressively in the L1-page TLBs compared to

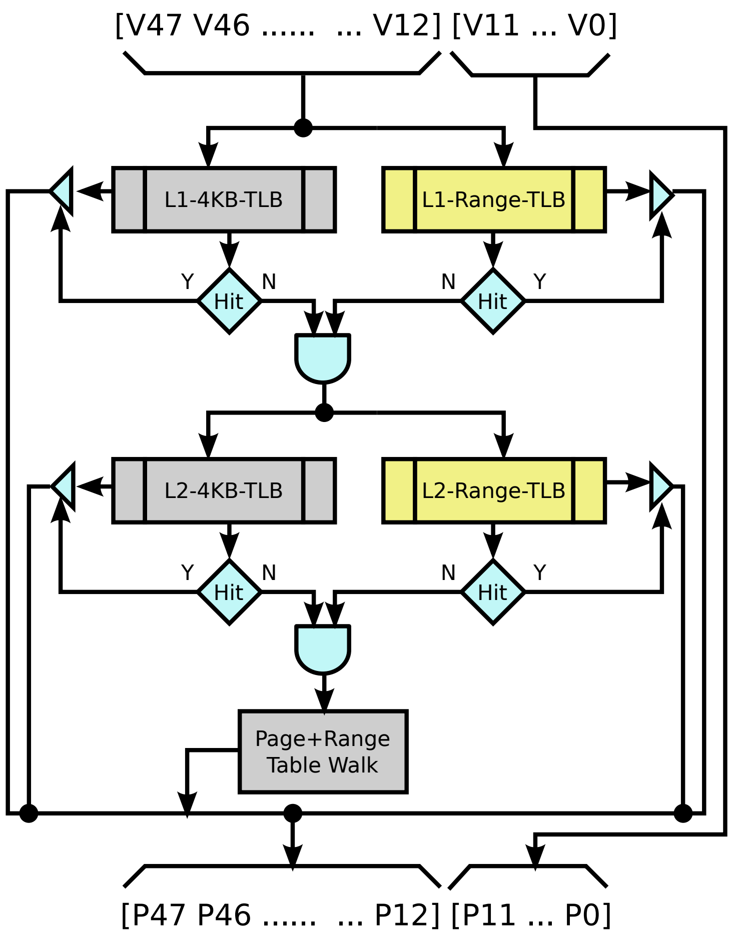


Figure 5.8: RMM$_{Lite}$ introduces an L1-range TLB and Lite (not shown) to the L1-page TLBs, in addition to the architectural support of RMM.

when only huge pages are supported, and reduces the total dynamic energy due to L1 TLB accesses.

Thus, RMM$_{Lite}$ makes the case for energy-efficient address translation, reducing further the energy overheads at all levels while improving also performance.

5.4.4 Discussion

Fully associative TLBs. We described Lite in the context of TLB organizations that support huge pages with separate set associative L1 TLBs [56], where each L1 TLB holds mappings of a single size. A different approach for huge page support is having a single fully associative L1 TLB that holds mappings of all page sizes [14, 107]. The same Lite mechanism for reducing dynamic energy applies to such TLB organizations. Although there is no notion of ways in a fully associative TLB, Lite clusters the distance of TLB hits from the LRU position as if there were ways, and reduces the TLB size in powers-of-two.

Lite's Cost. We did not analyze additional circuitry overheads for Lite, because the cost of computing the LRU distance and incrementing the corresponding counter (on a TLB hit) should be much lower than looking up the address (independently of a hit or miss) [47]. In addition, when the TLB operates with the minimum configuration (e.g., with only 1-way active), the additional circuits of Lite are not used and do not affect dynamic energy; this case is responsible for 63.7% of L1 TLB lookups with RMM$_{Lite}$ (Table 5.5).

5.5 Methodology

This section describes our simulation infrastructure and benchmarks.

Simulation infrastructure. We developed a Memory Management Unit (MMU) simulator based on Pin [88], instrument all memory operations, and simulate various TLB configurations. Because TLB studies require longer instruction counts than other processor components for applications to realistically stress the TLBs, slow cycle-accurate simulators make for infeasibly long simulation times. Thus, we developed our own simulation infrastructure that focuses on the address translation path based on Pin, `pagemap`, and Cacti. Note that we avoid using BadgerTrap [51] here because we need to instrument all memory operations and simulate L1 TLB accesses, while BadgerTrap allows instrumenting only the L2 TLB misses of the host machine, as explained in Section 4.7.

For a simulated L2 TLB miss, we access the real page table of the running process through `pagemap` [10] to determine whether it is a 4 KB page, a 2 MB page, or a range translation entry and its boundaries. Our simulation infrastructure and the simulated applications run on a host with Linux kernel v3.15.5. To deduce the number of required memory references per page walk, we simulate a per-core MMU cache based on Intel's Paging Structure Caches [61]. The MMU cache consists of three individual structures, each of which holds different levels of the page table (PDE, PDPTE, and PML4 levels). These structures are all accessed in parallel after an L2 TLB miss. The configuration details of the MMU cache is based on [28] and summarized in Table 5.2.

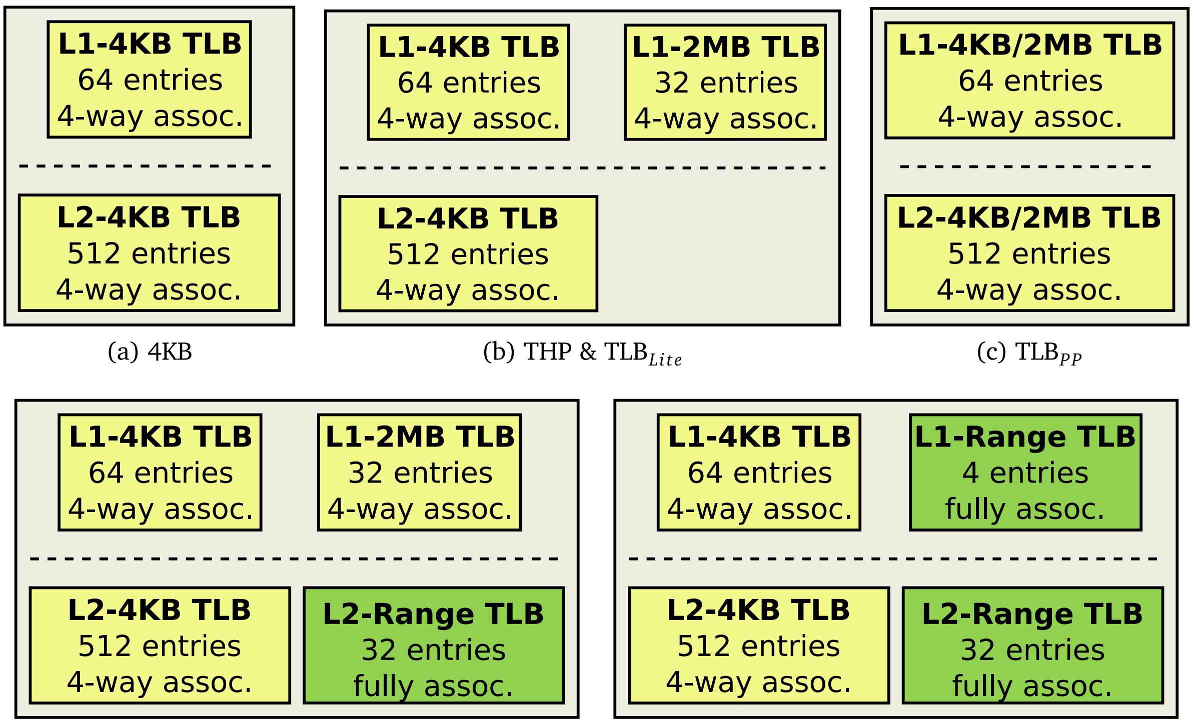
Component	Size (entr.)	Assoc.	Read (pJ)	Write (pJ)	Leakage (mW)
L1-4KB TLB	64	4-way	5.865	6.858	0.3632
	32	2-way	1.881	2.377	0.1491
	16	1-way	0.697	0.945	0.0636
L1-2MB TLB	32	4-way	4.801	5.562	0.1715
	16	2-way	1.536	1.924	0.0703
	8	1-way	0.568	0.764	0.0295
L1-range TLB	4	fully	1.806	1.172	0.1395
L2-4KB TLB	512	4-way	8.078	12.379	1.6663
L2-range TLB	32	fully	3.306	1.568	0.2401
MMU-cache$_{PDE}$	32	2-way	1.824	2.281	0.1402
MMU-cache$_{PDPTE}$	4	fully	0.766	0.279	0.0500
MMU-cache$_{PML4}$	2	fully	0.473	0.158	0.0296
L1-Cache	32KB	8-way	174.171	186.723	13.3364

Table 5.2: Dynamic energy per read operation and write operation, and leakage power with 32 nm process technology for the memory structures that participate in address translation, based on Cacti [83].

We use Cacti [83] with 32 nm process technology to estimate the dynamic energy of the memory structures that participate in address translation. To estimate the dynamic energy of an N-entry range TLB, we use Cacti with the configuration of a regular fully associative page TLB, but with 2× more tag bits in order to account for the effect of the double comparison that takes place in the range TLB. To estimate the dynamic energy of a page TLB with some ways disabled (e.g., 64-entry 4-way, with 2 ways disabled), we use the dynamic energy results from Cacti for the resulting smaller structure (e.g., 32-entry 2-way TLB). Table 5.2 summarizes the results from Cacti for all simulated structures.

We couple the results from our MMU simulator with those from Cacti. This simulation infrastructure computes misses and hits per memory structure, and estimates the dynamic energy and the cycles spent in L1 and L2 TLB misses.

Configurations. We simulate the following configurations: (i) *4KB* supports only 4 KB pages. (ii) *THP* supports 4 KB and 2 MB pages through transparent huge pages [5], and is the state of the practice for reducing the performance overhead of L1 and L2 TLB misses. (iii) TLB_{Lite} uses the same TLB organization with THP, but also includes the Lite mechanism in the L1-4KB TLB and L1-2MB TLB. (iv) *RMM* supports 4 KB, 2 MB pages, and an L2-range



(a) 4KB (b) THP & TLB$_{Lite}$ (c) TLB$_{PP}$

(d) RMM (e) RMM$_{Lite}$

Figure 5.9: TLB configurations.

TLB. (v) *TLB$_{PP}$* is a perfect implementation of TLB$_{Pred}$ [95]. TLB$_{Pred}$ is a state of the art scheme that seeks to improve the energy efficiency of TLBs by supporting different page sizes in a single set associative TLB. TLB$_{Pred}$ uses prediction to decide whether a reference goes to a huge page or not, in order to choose the appropriate TLB index bits and access the TLB. Our perfect implementation of TLB$_{Pred}$, named TLB$_{PP}$, assumes a perfect predictor with no energy overhead that always chooses the correct page size per lookup operation. In addition, TLB$_{PP}$ mixes 4 KB and 2 MB pages in both L1 and L2 TLBs. (vi) *RMM$_{Lite}$* supports 4 KB pages and range translations in both L1 and L2 TLBs, and includes the Lite mechanism in the L1-4KB TLB.

We set the threshold ϵ of Lite to 12.5% (1/8$_{th}$) relative increase in MPKI for *TLB$_{Lite}$* and as a 0.1 absolute increase for *RMM$_{Lite}$*, i.e., Lite disables ways if the predicted MPKI remains less than 12.5% or by 0.1 compared to the MPKI with fully enabled resources for *TLB$_{Lite}$* and *RMM$_{Lite}$*. Lite reduces the L1 TLBs down to 1-way active but never turns off completely an L1 TLB in our experiments. Finally, *RMM* and *RMM$_{Lite}$* use perfect eager paging, i.e.,

Energy Model	
TLBs / MMUcache	$E_{TLB/MMUcache} = A * E_{read} + M * E_{write}$
Page walks	$E_{page\_walks} = Mem * Eread_{L1\_cache}$
Total energy	$E_{total} = \sum_{i=1}^{n}(E_{TLBi/MMUcachei}) + E_{page\_walks}$
Performance Model	
L1 TLB hits	$Cycles_{L1TLBmisses} = 0$ (all L1 TLBs are accessed in parallel with L1 dcache)
L1 TLB misses	$Cycles_{L2TLBmisses} = M_{L1TLBs} * 7$ (all L2 TLBs are accessed in parallel)
L2 TLB misses	$Cycles_{L2TLBmisses} = M_{L2TLBs} * 50$
Total cycles	$Cycles_{TLBmisses} = Cycles_{L1TLBmisses} + Cycles_{L1TLBmisses}$
A: Accesses	M: Misses
Mem: Memory references to fetch PTEs (up to 4)	

Table 5.3: Dynamic energy and performance models.

the operating system perfectly allocates all contiguous pages of virtual address space to contiguous physical pages. Our actual eager paging implementation generates few and large range translations that map most of a process's address space for applications that run natively in Linux, as described and shown in Chapter 4. However, this is not always the case for our actual eager paging implementation with applications that run on top of Pin because: (i) we had to slightly modify eager paging so that Pin-based applications run properly at the cost of allocating less memory requests eagerly, (ii) we could not use the TCMalloc library that further increases the contiguity in range translations because some Pin-based applications were crashing, and (iii) because Pin introduces an extra level of indirection between the application and the operating system, including internal memory management operations, that further affect the quality of ranges. Since in this chapter we focus on micro-architectural techniques that improve the energy-efficiency of address translation, we hence abstract away the allocation support for range translations. However, we provide an analysis of the impact of eager paging on energy and performance at the end of Section 5.6. Figure 5.9 summarizes the simulated configurations and the corresponding parameters.

Suite	Description	Application	Memory
SPEC 2006	compute & memory intensive single-threaded workloads	astar	350 MB
		cactusADM	690 MB
		GemsFDTD	860 MB
		mcf	1.7 GB
		omnetpp	165 MB
		zeusmp	530 MB
PARSEC	RMS multi-threaded workloads	canneal	780 MB
BioBench	Bioinformatics single-threaded workloads	mummer	470 MB

Table 5.4: Workload description and memory footprint.

Dynamic energy model. We report the amount of dynamic energy spent in the address translation path. Table 5.3 summarizes the equations of our energy model. The dynamic energy per translation structure is the sum of the dynamic energy spent due to lookup operations and the dynamic energy spent due to write operations after misses. Our model for the dynamic energy spent in page walks optimistically assumes that all page walk references always hit in the L1 cache of the memory hierarchy.

Performance model. We report misses per thousand instructions in the L1 and L2 TLBs, and cycles spent in L1 and L2 TLB misses. Our estimations are based on the following assumptions: (i) L1 TLBs are accessed in parallel with the data cache, so L1 TLB hits add no cycles, (ii) L1 TLB misses trigger lookup accesses in L2 TLBs that take 7 cycles [63], and (iii) L2 TLB misses trigger page walks that take 50 cycles [77] for all applications. Thus, the cycles spent in TLB misses are the sum of the cycles spent in L1 TLB misses and in L2 TLB misses. Table 5.3 summarizes the equations of our performance model. Note that short L1 TLB misses, like those that hit in the L2 TLB, can be overlapped with execution in some cases, and may not decrease performance by that much. For RMM and RMM$_{Lite}$, the range table walks occur in the background and do not add to the execution time, but they incur dynamic energy overhead.

Benchmarks. We focus on various workloads that exhibit poor TLB performance from Spec2006 [59], BioBench [15], and Parsec [33], summarized in Table 5.4. We define as TLB intensive workloads those that experience more than 5 L1 TLB misses per thousand instructions with 4 KB pages. We also report results for all remaining Spec2006 and Parsec workloads in the sensitivity analysis subsection. We fast-forward the execution for 50 billion instructions, and then simulate for the next 50 billion instructions.

5.6 Results

This section evaluates the two proposed TLB organizations: TLB_{Lite} that adds the Lite mechanism on top of TLB support for huge pages, and RMM_{Lite} that adds the Lite mechanism and the 4-entry L1-range TLB on top of RMM. We first evaluate how these TLB organizations reduce the dynamic energy in address translation and the cycles spent in L1 and L2 TLB misses for a set of TLB intensive workloads. Then we present results for more workloads, and finally we perform a sensitivity analysis based on the interval size and the random probability of Lite.

5.6.1 Dynamic Energy & Performance

Figure 5.10 shows the reduction of the dynamic energy in address translation and the cycles spent in L1 and L2 TLB misses for all the simulated configurations explained in Section 5.5. The results are normalized to the 4KB configuration.

Overview. The results show that (i) TLB_{Lite} reduces the dynamic energy with respect to THP (Figure 5.10 top) without significantly affecting the performance (Figure 5.10 bottom), and (ii) RMM_{Lite} further reduces the dynamic energy of TLB lookups and eliminates almost completely the performance and the associated energy overheads of L1 TLB misses.

4KB exhibits two sources of dynamic energy overhead: the L1 TLB lookups and the page walks. Depending on the workload's locality in the TLB hierarchy, one of the two sources dominates. Figure 5.11 shows the MPKI for the L1 and L2 TLBs. The L1 TLB lookups are responsible for the majority of overhead in these workloads, except for cactusADM and mcf that suffer more frequently from page walks. In addition, previous studies have shown that 4 KB pages lead to significant performance overhead [27, 28, 77] that increases in turn the total static energy.

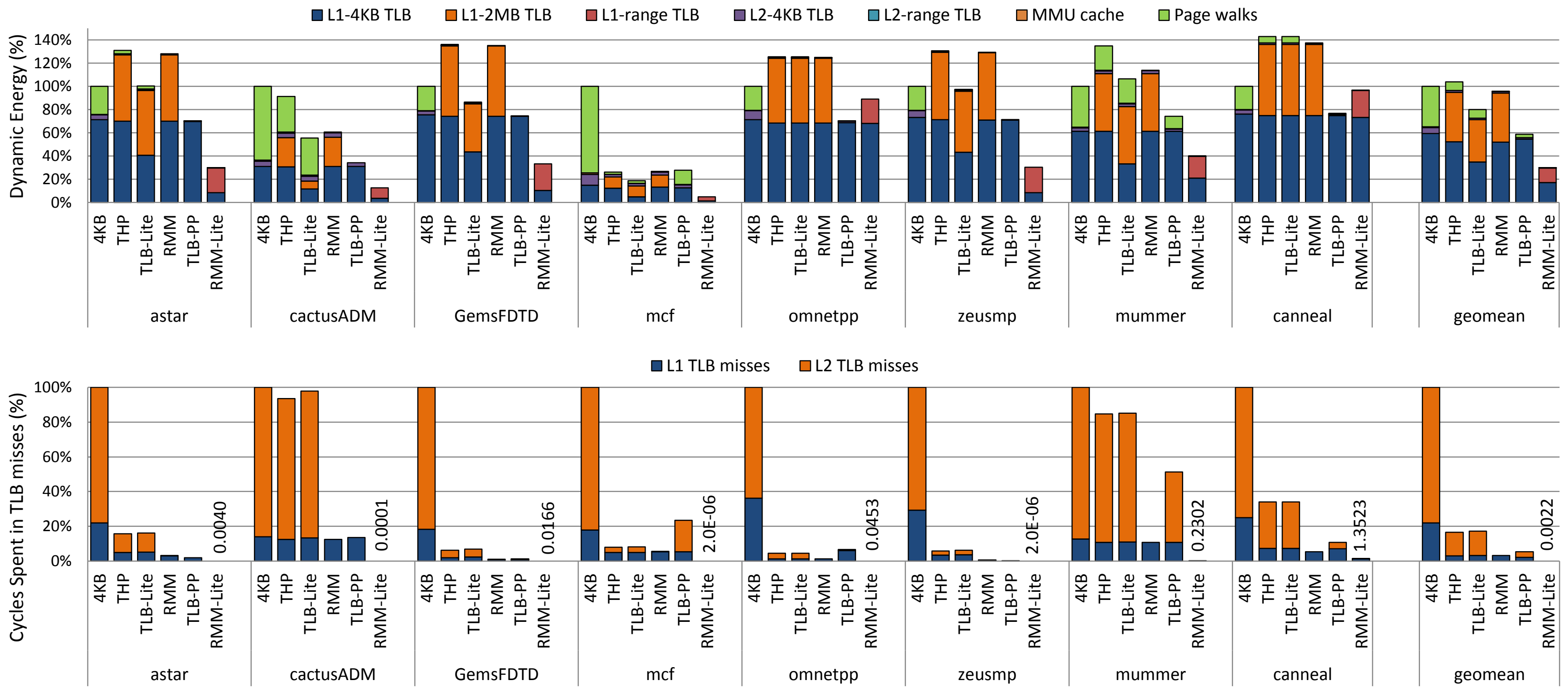


Figure 5.10: Dynamic energy spent in address translation and cycles spent in TLB misses for TLB intensive workloads.

THP reduces significantly the portion of dynamic energy and the performance overhead of page walks, due to fewer L1 and L2 TLB misses. However, THP increases the amount of dynamic energy spent in the L1 TLBs because the L1-2MB TLB is accessed on every memory operation, in addition to the L1-4KB TLB. These accesses increase the total dynamic energy consumption compared to 4KB for most workloads—up to 43% for canneal. On average, THP increases the dynamic energy by 4%, while reducing the cycles spent in TLB misses by 83%, compared to 4KB pages.

TLB$_{Lite}$ reduces the dynamic energy by 23% on average and by 40% and 37% for cactusADM and GemsFDTD, compared to THP. TLB$_{Lite}$ opportunistically reduces the dynamic energy spent in address translation when the utility of having all ways active becomes low. Table 5.5 shows the percentage of active ways during the execution time. On average, all 4-ways are active for 51% and 81% of the time in the L1-4KB TLB and L1-2MB TLB. In addition, TLB$_{Lite}$ barely affects performance for most workloads except for canneal. Compared to the THP configuration, TLB$_{Lite}$ increases the L1 and L2 TLB misses by 4% and 3% on average, and the cycles spent in TLB misses from 16.6% to 17.2%. Note that cycles spent in short TLB misses may be overlapped with execution; thus the impact on total execution time is likely to be lower.

RMM eliminates the dynamic energy and performance overheads of page walks due to the L2-range TLB. However, the dynamic energy spent in the L1 TLBs remains high, similar to that with THP, because of accessing both L1 TLBs for 4KB and 2MB pages. On average, RMM reduces the dynamic energy by only 8% and the cycles spent in TLB misses by 80%, compared to THP.

TLB$_{PP}$ is a perfect implementation of TLB$_{Pred}$ [95], as explained in Section 5.5. We observe that TLB$_{PP}$ reduces the dynamic energy and performance overheads of page walks because it enjoys larger reach compared to THP. In addition, the TLB$_{PP}$ reduces the dynamic energy in L1 TLBs since only a single structure for both 4 KB and 2 MB pages is accessed on every memory operation, but these results under report its true costs. On average, TLB$_{PP}$ would reduce the dynamic energy by 43% and the cycles spent in TLB misses by 67% compared to THP, but is unrealizable in practice.

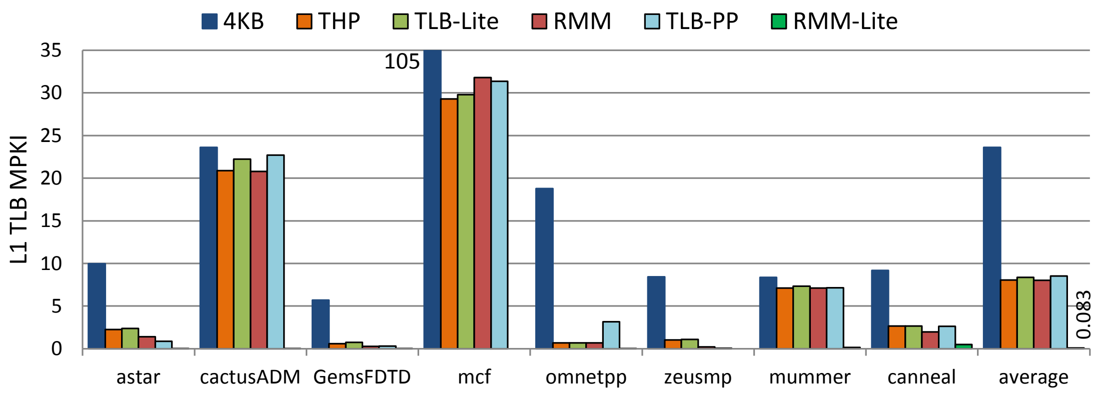
RMM$_{Lite}$ reduces the dynamic energy in address translation the most compared to the other approaches. RMM$_{Lite}$ reduces dynamic energy by more than 80% for mcf and cactusADM, and by 71% on average while eliminating more than 99% of cycles spent in TLB misses compared to THP. This occurs because the high hit ratio of the L1-range TLB allows

Benchmark	**TLB$_{Lite}$**						**RMM$_{Lite}$**		
	L1-4KB TLB			**L1-2MB TLB**			**L1-4KB TLB**		
	4-ways	**2-ways**	**1-way**	**4-ways**	**2-ways**	**1-way**	**4-ways**	**2-ways**	**1-way**
astar	39.6	57.2	3.2	96.7	3.3	0.0	0.0	0.1	99.9
cactusADM	22.8	24.0	53.2	14.6	11.9	73.5	0.1	0.1	99.9
GemsFDTD	42.9	44.9	12.2	54.4	41.7	4.0	2.3	0.4	97.4
mcf	25.8	26.7	47.5	97.8	1.7	0.5	0.0	0.0	100.0
omnetpp	100.0	0.0	0.0	100.0	0.0	0.0	99.3	0.7	0.0
zeusmp	45.5	43.5	11.1	86.6	13.3	0.1	0.0	0.0	100.0
mummer	32.8	67.2	0.0	98.4	0.5	1.0	7.8	79.4	12.9
canneal	100.0	0.0	0.0	100.0	0.0	0.0	97.5	2.5	0.0
average	**51.2%**	**32.9%**	**15.9%**	**81.1%**	**9.0%**	**9.9%**	**25.9%**	**10.4%**	**63.7%**

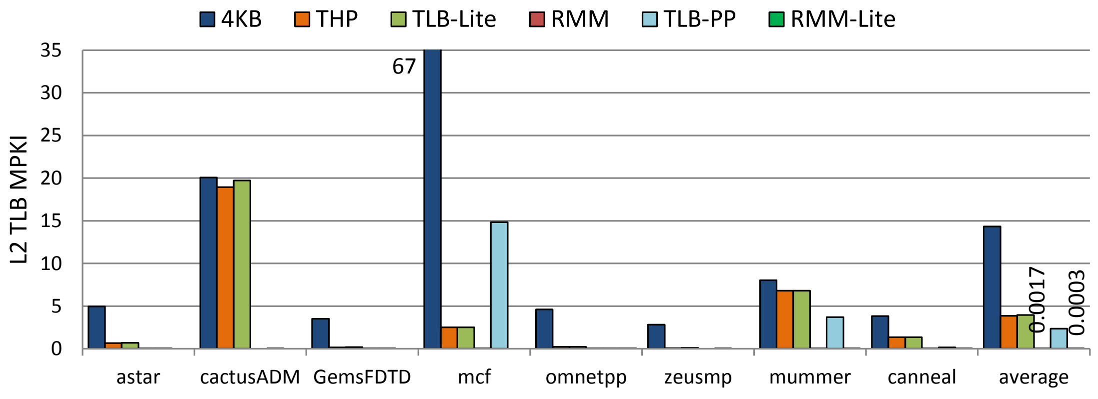
Table 5.5: Percentage of lookups with 4, 2 and 1 active ways in the L1-page TLBs for TLB$_{Lite}$ and RMM$_{Lite}$. RMM$_{Lite}$ disables more ways than TLB$_{Lite}$ thanks to the high hit ratio of the L1-range TLB.

	TLB$_{Lite}$		**RMM$_{Lite}$**	
Benchmark	**L1-4KB**	**L1-2MB**	**L1-4KB**	**L1-2MB**
astar	75.7	24.3	32.4	67.6
cactusADM	90.8	9.2	0.0	100.0
GemsFDTD	30.1	69.9	0.1	99.9
mcf	38.9	61.1	12.0	88.0
omnetpp	55.2	44.8	51.0	49.0
zeusmp	37.6	62.4	0.0	100.0
mummer	95.7	4.3	5.8	94.2
canneal	91.0	9.0	25.9	74.1
average	**64.4%**	**35.6%**	**15.9%**	**84.1%**

Table 5.6: Percentage of hits in the L1 TLBs for TLB$_{Lite}$ and RMM$_{Lite}$.



(a) L1 TLB MPKI



(b) L2 TLB MPKI

Figure 5.11: L1 and L2 TLB misses per thousand instructions.

Lite to disable ways more aggressively in the L1-4KB TLB. Table 5.6 shows the percentage of L1 TLB hits that come from the L1-4KB TLB and the L1-range TLB. The L1-range TLB contributes by 84.1% to the L1 TLB hits, and thus, RMM$_{Lite}$ depends less on the performance of the L1-4KB TLB and runs 63.7% of the time with only 1-way active in the L1-4KB TLB (Table 5.5).

Compared to TLB$_{PP}$, RMM$_{Lite}$ brings less dynamic energy improvements only for omnetpp and canneal because the L1-4KB TLB has high utilization for those workloads (Table 5.5). Still, RMM$_{Lite}$ reduces the dynamic energy overhead by 49% on average, compared to TLB$_{PP}$. Note that RMM$_{Lite}$ and TLB$_{PP}$ are orthogonal; a combined approach could use the L1-range TLB for range translations, the TLB$_{PP}$ for pages, and the Lite mechanism to disable ways opportunistically, as with regular page TLBs.

In addition to the dynamic energy savings, RMM$_{Lite}$ significantly reduces L1 and L2 TLB misses, further improving the performance and reducing static energy overheads. Compared to RMM, RMM$_{Lite}$ improves performance more because it eliminates most L1 TLB misses, in addition to eliminating most L2 TLB misses as RMM does. Overall RMM$_{Lite}$ makes a good case for energy-efficient address translation.

5.6.2 Sensitivity Analysis

Other workloads. Our evaluation in the previous section focused on a set of TLB intensive workloads. For completeness, we ran experiments with other workloads that stress the TLB hierarchy less and observed similar results. Figure 5.12 shows the reduction in dynamic energy for the rest of Spec2006 (top and middle) and Parsec (bottom) workloads. On average, TLB$_{Lite}$ reduces the dynamic energy spent in address translation by 26% and 20% for those Spec2006 and Parsec workloads, while RMM$_{Lite}$ reduces the dynamic energy by 72% and 66%. Regarding performance, the results are similar to those for the TLB intensive workloads.

Interval size and random probability. Lite depends on the size of the interval and the random probability for activating all ways in the L1 TLBs. To quantify the impact of these parameters, we performed a sensitivity analysis varying the interval size from 1 million to 10 million instructions and the random probability from 1/8 to 1/128. We find that Lite performs slightly better in terms of both performance and dynamic energy, with shorter interval and with lower probability. The short interval allows Lite to respond faster to performance changes, while the low probability avoids frequently enabling all ways to check the potential for performance improvement.

Impact of Eager Paging. Our results for RMM and RMM$_{Lite}$ in the previous sections of this chapter are based on perfect eager paging, i.e., the operating system perfectly allocates all contiguous pages of virtual address space to contiguous physical pages. The reasons for using perfect eager paging are summarized in Section 5.5.

In this section we want to quantify the impact of eager paging on energy and performance for RMM$_{Lite}$. Figure 5.13 shows the dynamic energy spent in address translation and the cycles spent in L1 and L2 TLB for RMM$_{Lite}$ with perfect eager paging (RMM$_{Lite-PP}$), and with our actual implementation of eager paging (RMM$_{Lite-EP}$), that was presented in Chapter 4 but with the limitations that are described in Section 5.5. For these experiments

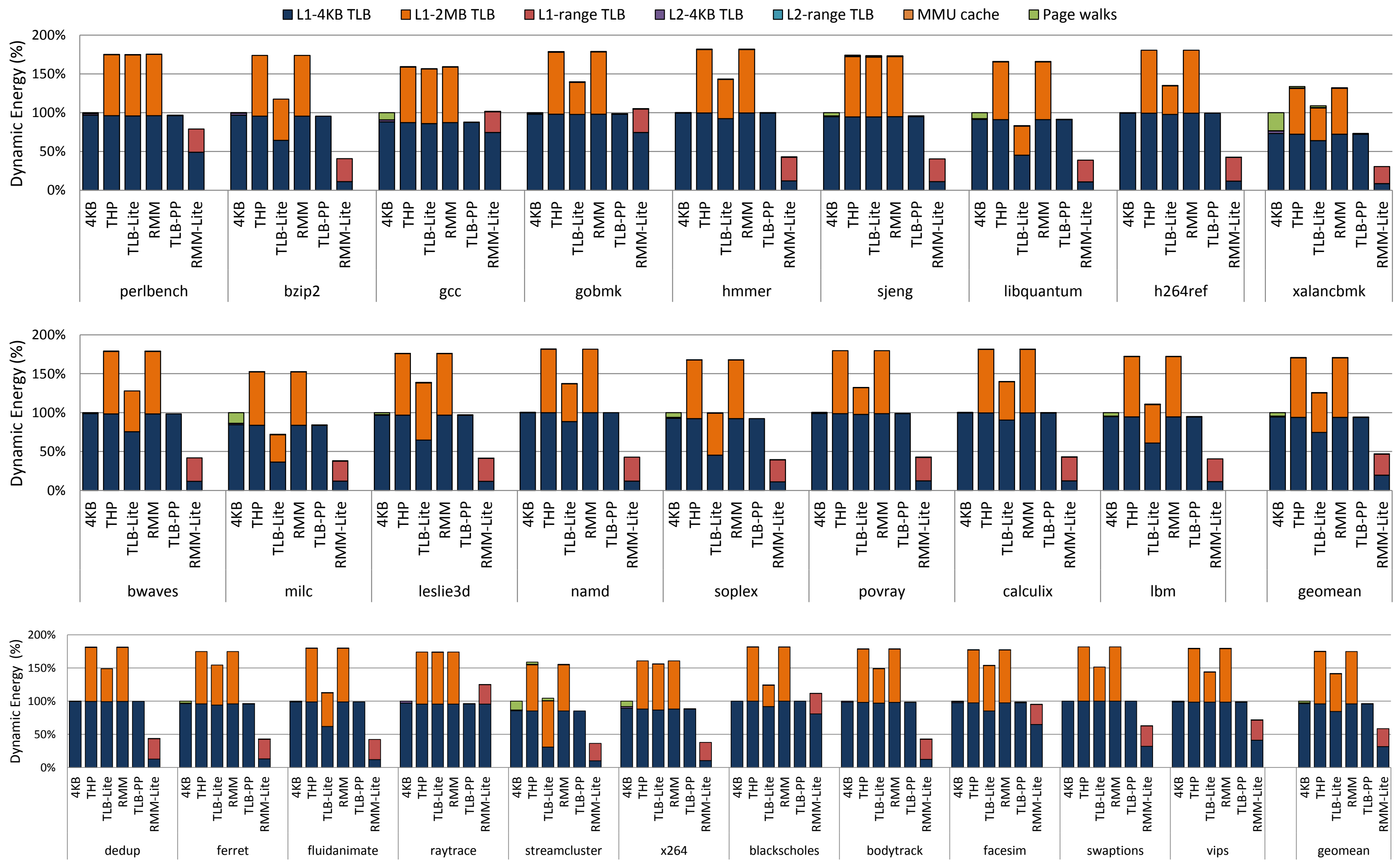


Figure 5.12: Dynamic energy reduction for the rest of Spec2006 (top and middle) and Parsec (bottom) workloads.

we fast forward for 50 billion instructions and then simulate for the next 1 billion instructions. The results are normalized to the 4KB configuration.

We observe that RMM$_{Lite-EP}$ performs close to RMM$_{Lite-PP}$ for most workloads (e.g., mcf, omnetpp, soplex, mummer, canneal), reducing by similar ratio the dynamic energy spent in address translation and the cycles spent in L1 and L2 TLB misses. However, the savings differ between RMM$_{Lite-EP}$ and RMM$_{Lite-PP}$ for astar, cactusADM, and GemsFDTD. The reason is that our current implementation of eager paging generates less contiguity in range translations for Pin-based applications than perfect eager paging does for the reasons explained in Section 5.5. This reduced contiguity affects correspondingly the energy and performance savings.

Overall, these results show that even a less sophisticated implementation of eager paging can help in reducing the energy and performance overheads of address translation with RMM$_{Lite}$. However, these results indicate also the need for enhancements in the implementation of eager paging, and more generally in allocating memory for range translations, to render the benefits of range translations more applicable to more workloads. Such enhancements could be the subject of future work.

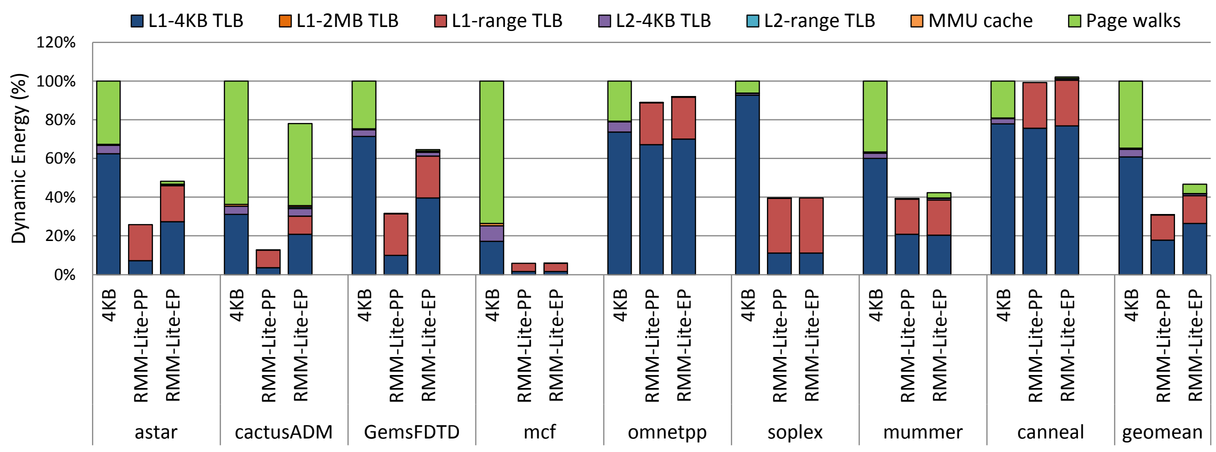
Reducing static energy. Although we focused on reducing the dynamic energy of address translation, the proposed techniques can also reduce the static (leakage) energy of TLBs when combined with schemes that power-gate the disabled ways [57, 99].

Threshold. The benefits of Lite depend also on the threshold ϵ for increased MPKI due to way-disabling. The threshold choice introduces a trade-off between dynamic and static energy. Studying the impact of different thresholds on total energy and performance could be the subject of future work.

5.7 Related Work

This section reviews the related work (except for TLB$_{Pred}$ [95] discussed in Section 5.6), categorized into techniques that optimize TLBs for energy efficiency, dynamically resizing TLBs, selective TLB lookups, and virtual caches.

Optimizing TLBs for energy efficiency. Several techniques have been proposed to improve the energy efficiency of TLBs. Juan et al. [71] proposed circuit optimizations that



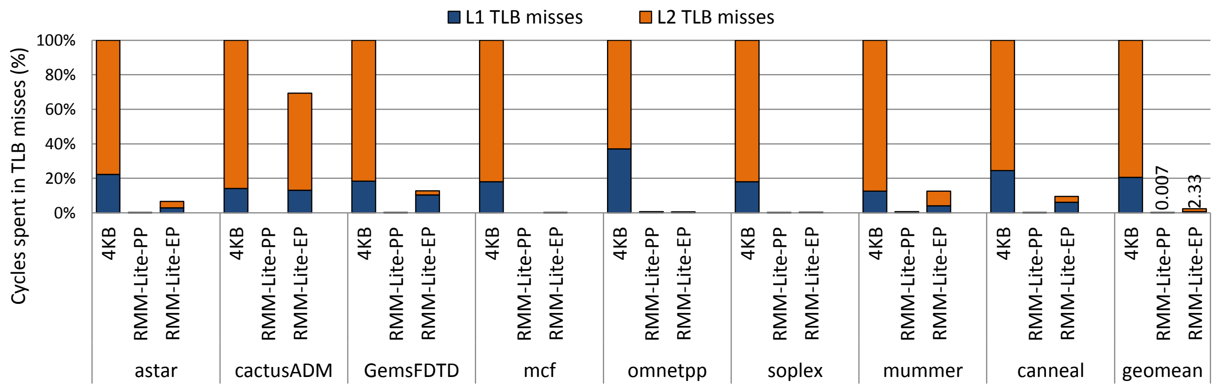


Figure 5.13: Dynamic energy spent in address translation and cycles spent in TLB misses for RMM$_{Lite}$ with perfect eager paging (RMM$_{Lite-PP}$) and with our actual eager paging implementation (RMM$_{Lite-EP}$).

reduce the lookup energy in TLBs. Banked TLBs [38, 41, 82] and TLB filtering [22, 38, 49] can also help in reducing dynamic energy by accessing only one bank or just a filter on each memory operation. Similarly, Lee et al. [82] proposed a partitioned L1 TLB, with each part serving translations for a semantic region (stack, heap, global data). That TLB organization was further improved leveraging the low entropy of information in the stack and global data memory addresses [21]. To reduce the TLB energy for multi-issue superscalar processors, Ballapuram et al. [22] proposed a compaction mechanism for issuing only a single TLB lookup, when multiple memory references access the same page at the same cycle. Xue et al. [119] proposed to speculatively perform address translation, based on the base-displacement address, by accessing a small L0 TLB early in the pipeline, so that the translation latency is not increased. Finally, Seyedi et al. [105] proposed combining nano electro mechanical switches with CMOS technology for fully associative L1 TLBs.

While these techniques reduce the dynamic energy spent in TLBs, they do not consider mechanisms that increase TLB reach [27, 51, 78, 96, 97, 111] to improve energy efficiency. Our proposed designs leverage the benefits of such mechanisms to reduce the total energy spent in address translations. Thus, TLB$_{Lite}$ and RMM$_{Lite}$ are orthogonal to those approaches, and could further improve their benefits.

Dynamically resizing TLBs. Balasubramonian et al. [20] proposed an interval-based scheme to dynamically resize the TLB, trading off dynamic energy for performance. The objective of that approach is similar to Lite. However, their design and algorithm targets a monolithic, fully associative TLB and tracks only whether a TLB entry was referenced or not to decide for resizing. Thus, the energy savings opportunity becomes lower in case that the TLB entries are referenced only few times but not heavily utilized. In contrast, Lite tracks the utility of TLB entries in the miss ratio, considers the presence of separate L1 TLBs, and provides better opportunity for resizing TLB resources.

Selective lookups in TLBs. Kadayif et al. [72] combined hardware and compiler techniques to avoid lookups in instruction TLBs. A register holds the most recently used iTLB entry, and the compiler generates instructions that access only a register instead of the iTLB. That approach was extended later for the data TLB [73, 74]. However, the TLBs are still used in such system. Thus, TLB$_{Lite}$ and RMM$_{Lite}$ are again orthogonal and can further reduce the total energy cost of address translation in these systems.

Virtual caches. Prior work proposed virtual caches [26, 66, 116] to reduce the energy and performance overheads of address translation. With virtual caches, the cache hierarchy is accessed without TLB lookups, unless a cache miss occurs. While saving almost all TLB energy, they introduce many more changes to the architecture and require additional support to handle synonyms and enforce protection.

5.8 Summary

The goal of this chapter was to improve the energy efficiency in address translation. We proposed *Lite*, a mechanism that monitors the performance and utility of L1 TLBs and adaptively changes their sizes with way-disabling, and applied Lite to a standard TLB hierarchy with support for huge pages, named TLB$_{Lite}$. In addition, we proposed RMM$_{Lite}$, based on Redundant Memory Mappings (RMM). RMM$_{Lite}$ augments RMM with an L1-range TLB and the Lite mechanism. The high hit ratio of the L1-range TLB allows Lite to disable ways in L1-page TLBs more aggressively. Our results show that TLB$_{Lite}$ reduces the dynamic energy spent in address translation by 23% with minimal impact on TLB miss cycles. RMM$_{Lite}$ further reduces the energy spent in address translation by 71% and the overhead from L1 TLB misses by 99%, on top of the near-zero L2 TLB misses of RMM.

6 Conclusions

This thesis analyzes the performance and energy overheads of virtual memory in address translation and proposes techniques to reduce them significantly.

We introduced the key concept of range translations. A range translation is a mapping between contiguous virtual pages mapped to contiguous physical pages with uniform protection. Range translations enable an efficient alternative representation of virtual to physical mappings to perform address translation, complementary to paging. We showed how range translations can improve the performance and energy efficiency of address translation while retaining the benefits of paging. We believe that range translations is the next logical step in the evolution of virtual memory.

6.1 Broader Impact

The limited TLB reach is a well-known problem to both acedemia and industry. Vendors have been enhancing the TLB resources in every processor generation, mainly through increasing hardware support for huge pages, to reduce the overhead of page walks. However, we believe that this approach falls short: as memory sizes increase more aggressively than TLB sizes, the virtual memory overheads that manifest in today's systems with 4 KB pages, will manifest similarly in tomorrow's systems with huge pages. Our experiments in this thesis show that such cases already exist, and our proposed solutions increase TLB reach to match the sizes of ever-growing memories.

In addition, energy efficiency has become an important parameter in all computing domains. To optimize a system for energy-efficiency, all involved components need to be addressed, including TLBs. Furthermore, TLBs are a well known source of power and energy in processors. Our proposed solutions improve the energy-efficiency of the address translation process and put another piece in solving the energy puzzle.

Finally, during the last years the interest and demand for fast and energy-efficient in-memory computing have increased extremely. In-memory computing leverages the ever-increasing amount of available physical memory to provide adequate support for low latency and real time applications that operate on huge data sets. However, if all data reside in memory, then the address translation plays an even more important role for accessing memory. The contributions of this thesis enable fast and energy-efficient address translation and pave the way for efficient in-memory computing.

6.2 Future Research Directions

Some of the contributions described in this thesis may be further extended. In this section we provide suggestions for future research directions.

Analyzing Address Translation Overheads. Our work on quantifying the performance overhead of address translation for memory intensive workloads opens new directions for further research. An interesting direction would be to investigate the performance cost of address translation on newer processors that provide better TLB support with more entries, and measure how the costs change across the generations. In addition, newer processors offer on-chip energy counters that allow measuring directly the energy burnt by the processor. The use of such counters would allow to further understand the energy implications of address translation on a real system under complex long-running workloads. Finally, characterizing and analyzing the cost of emerging memory-intensive workloads under virtualized execution is another interesting direction for extending our work.

Towards Range-based Virtual Memory. Our work on range translations opens new opportunities and research challenges. We discussed some hardware and operating systems issues that a production implementation with range translations should consider. Further research is needed on memory management policies, that switch between eager and demand paging, decide when to form or break range translations, and integrate smoothly range translations with page-based mechanisms, such as memory compaction and defragmentation daemons. In addition, our work points towards the need of a contiguity aware memory allocator to replace the age-old buddy allocator used to manage physical memory.

Virtualizing range translations. Virtualizing range translations is also an interesting challenge with great potential for performance benefits. The high performance overhead of paging in virtualized environments is a well-established problem. As with virtualizing regular pages introduces various design options, such as nested paging and shadow paging, and requires adequate hardware support, virtualizing range translations requires a rigorous design space exploration of both software and hardware components to analyze the available choices, benefits, and costs. Taking into account the paramount importance of virtualization in cloud computing, we believe that range translations could play an important role in reducing virtualization overheads.

6.3 Further Acknowledgements

We want to credit the contributions of other students to this thesis. Jayneel Gandhi, from University of Wisconsin-Madison, contributed to the RMM design and evaluation in Chapter 4. Furkan Ayar, from Dumlupinar University and now at Yildiz Technical University, helped with the implementation of eager paging. Oriol Arcas and Ivan Ratkovic, from Barcelona Supercomputing Center, provided the Bluespec implementation and the synthesis results of the range TLB in Chapter 4.

This thesis has been supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain under the FPU National Plan (FPU12/04110), by the European Union (FEDER funds) under contracts TIN2007-60625, TIN2008-02055-E, TIN2012-34557, and TIN2015-65316-P, and by the European Union's Seventh Framework Programme (FP7/2007- 2013) under the ParaDIME project (GA no. 318693).

7 Publications

The contents of this thesis led to the following publications:

Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrián Cristal, and Michael M. Swift, **Performance Analysis of the Memory Management Unit under Scale-out Workloads**, In Proceedings of the 2014 IEEE International Symposium on Workload Characterization (**IISWC 2014**).

Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Unsal, **Redundant Memory Mappings for Fast Access to Large Memories**, In Proceedings of the 42nd International Symposium on Computer Architecture (**ISCA 2015**) — ***Selected for IEEE Micro's "Top Picks from 2015 Computer Architecture Conferences"***.

Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal, **Energy-Efficient Address Translation**, In Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture (**HPCA 2016**).

Jayneel Gandhi, **Vasileios Karakostas**, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal, **Range Translations for Fast Virtual Memory**, In IEEE Micro Special Issue: Top Picks from 2015 Computer Architecture Conferences (**IEEE Micro Top Picks 2016**)

The following papers were also published while on graduate studies but are not included in or directly related to this thesis:

Gokcen Kestor, **Vasileios Karakostas**, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero, **RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems**, In Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering (**ICPE 2011 - Best Paper Award**).

Carlos Villavieja, **Vasileios Karakostas**, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrián Cristal, and Osman S. Unsal, **DiDi: Mitigating The Performance Impact of TLB Shootdowns Using A Shared TLB Directory**, In Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (**PACT 2011**).

Vasileios Karakostas, Saša Tomić, Osman S. Unsal, Mario Nemirovsky, and Adrián Cristal, **Improving the Energy Efficiency of Hardware-Assisted Watchpoint Systems**, In Proceedings of the 50th Design Automation Conference (**DAC 2013**).

Srdan Stipic, **Vasileios Karakostas**, Vesna Nowack, Adrián Cristal, Osman S. Unsal, and Mateo Valero, **Dynamic Transaction Coalesing**, In Proceedings of the 11th conference on ACM Computing Frontiers (**Computing Frontiers 2014**).

Azam Seyedi, **Vasileios Karakostas**, Stefan Cosemans, Adrián Cristal, Mario Nemirovsky, and Osman S. Unsal, **NEMsCAM: A Novel CAM Cell based on Nano-Electro-Mechanical Switch and CMOS for Energy Efficient TLBs**, In Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (**NANOARCH 2015**).

References

[1] Intel® Itanium® Architecture Developer's Manual, Vol. 2. Cited on page: 89

[2] Sh-3 RISC Processor family. Cited on page: 2, 4, 21, 93

[3] Intel Strongarm Processor. Cited on page: 2, 4, 21, 93

[4] Intel 8086 - Wikipedia. http://en.wikipedia.org/wiki/Intel_8086, Cited on page: 28, 90

[5] Transparent Huge Pages in 2.6.38. http://lwn.net/Articles/423584/, Cited on page: 6, 9, 23, 35, 40, 62, 63, 65, 66, 84, 89, 96, 97, 99, 100, 103, 113

[6] Cloudsuite Overview. http://parsa.epfl.ch/cloudsuite/overview.html, Cited on page: 33, 35

[7] International Technology Roadmap for Semiconductors: 2012. http://www.itrs.net/Links/2012ITRS/Home2012.htm, Cited on page: 58

[8] Huge Pages Part 1 (Introduction). http://lwn.net/Articles/374424/, Cited on page: 6, 23, 35, 62, 65, 84, 89, 97

[9] The /proc filesystem. www.kernel.org/doc/Documentation/filesystems/proc.txt, Cited on page: 37

[10] Pagemap, from the userspace perspective. https://www.kernel.org/doc/Documentation/vm/pagemap.txt, Cited on page: 95, 99, 112

[11] perf: Linux profiling with performance counters . https://perf.wiki.kernel.org/index.php/Main_Page, Cited on page: 35, 81

[12] SPEC CPU^{TM} 2006. https://www.spec.org/cpu2006/, Cited on page: 46

[13] TCMalloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html, Cited on page: 79

[14] Advance Micro Devices. *Software Optimization Guide for AMD Family 15h Processors*. Number 47414, Cited on page: 24, 93, 97, 111

[15] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce L. Jacob, Chau-Wen Tseng, and Donald Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '05, pages 2–9, 2005. DOI: 10.1109/ISPASS.2005.1430554. Cited on page: 46, 81, 96, 116

[16] David H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-32, pages 248–259, 1999. DOI: 10.1109/MICRO.1999.809463. Cited on page: 8, 95, 103, 109

[17] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 108–120, 1991. DOI: 10.1145/106972.106985. Cited on page: 59

[18] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 20:1–20:13, 2013. DOI: 10.1145/2523616.2523629. Cited on page: 33

[19] Vlastimil Babka and Petr Tůma. Investigating cache parameters of x86 family processors. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 77–96, 2009. DOI: 10.1007/978-3-540-93799-9_5. Cited on page: 55

[20] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance

in general-purpose processor architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-33, pages 245–257, 2000. DOI: 10.1145/360128.360153. Cited on page: 8, 94, 126

[21] Chinnakrishnan Ballapuram, Kiran Puttaswamy, Gabriel H. Loh, and Hsien-Hsin S. Lee. Entropy-based low power data tlb design. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 304–311, 2006. DOI: 10.1145/1176760.1176797. Cited on page: 8, 94, 125

[22] Chinnakrishnan S. Ballapuram, Hsien-Hsin S. Lee, and Milos Prvulovic. Synonymous Address Compaction for Energy Reduction in Data TLB. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 357–362, 2005. DOI: 10.1145/1077603.1077689. Cited on page: 94, 125

[23] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 48–59, 2010. DOI: 10.1145/1815961.1815970. Cited on page: 21, 45, 89

[24] Thomas W. Barr, Alan L. Cox, and Scott Rixner. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 307–318, 2011. DOI: 10.1145/2000064.2000101. Cited on page: 90

[25] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture, Cited on page: 33

[26] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, 2012. Cited on page: 27, 53, 90, 94, 126

[27] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th*

Annual International Symposium on Computer Architecture, ISCA '13, pages 237–248, 2013. DOI: 10.1145/2485922.2485943. Cited on page: 2, 4, 6, 8, 21, 32, 42, 53, 58, 59, 61, 62, 63, 65, 71, 76, 79, 80, 81, 82, 84, 89, 94, 95, 97, 103, 117, 126

[28] Abhishek Bhattacharjee. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 46, pages 383–394, 2013. DOI: 10.1145/2540708.2540741. Cited on page: 2, 4, 6, 21, 32, 53, 55, 57, 59, 61, 63, 80, 89, 94, 97, 112, 117

[29] Abhishek Bhattacharjee and Margaret 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. DOI: 10.1109/PACT.2009.26. Cited on page: 38, 52, 59, 89

[30] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core Cooperative TLB for Chip Multiprocessors. In *ASPLOS*, pages 359–370, DOI: 10.1145/1736020.1736060. Cited on page: 58

[31] Abhishek Bhattacharjee, Daniel Lustig, and Margaret 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. Cited on page: 89, 90

[32] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, 2011. AAI3445564. Cited on page: 46, 81

[33] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, 2008. DOI: 10.1145/1454115.1454128. Cited on page: 96, 116

[34] David L. Black, Richard F. Rashid, David B. Golub, and Charles R. Hill. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and*

Operating Systems, ASPLOS III, pages 113–122, 1989. DOI: 10.1145/70082.68193. Cited on page: 25, 74

[35] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, 2008. DOI: 10.1145/1375581.1375586. Cited on page: 79

[36] Michel Cekleov and Michel Dubois. Virtual-address caches part 1: Problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, DOI: 10.1109/40.621215. Cited on page: 27, 53

[37] Michel Cekleov and Michel Dubois. Virtual-address caches, part 2: Multiprocessor issues. *IEEE Micro*, 17(6):69–74, DOI: 10.1109/40.641599. Cited on page: 27, 53

[38] Yen-Jen Chang and Mao-Feng Lan. Two New Techniques Integrated for Energy-efficient TLB Design. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(1):13–23, DOI: 10.1109/TVLSI.2006.887813. Cited on page: 8, 94, 125

[39] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 114–123, 1992. DOI: 10.1145/139669.139708. Cited on page: 59

[40] Yu-Ting Chen, Jason Cong, Hui Huang, Bin Liu, Chunyue Liu, Miodrag Potkonjak, and Glenn Reinman. Dynamically Reconfigurable Hybrid Cache: An Energy-efficient Last-level Cache Design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 45–50, 2012. DOI: 10.1109/DATE.2012.6176431. Cited on page: 58

[41] Jin-Hyuck Choi, Jung-Hoon Lee, Seh-Woong Jeong, Shin-Dug Kim, and Charles C. Weems. A Low Power TLB Structure for Embedded Systems. *Computer Architecture Letters*, 1, DOI: 10.1109/L-CA.2002.1. Cited on page: 8, 94, 125

[42] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Trans. Comput. Syst.*, 3(1):31–62, DOI: 10.1145/214451.214455. Cited on page: 59

[43] Nachshon Cohen and Erez Petrank. Limitations of partial compaction: Towards practical bounds. *SIGPLAN Not.*, 48(6):309–320, DOI: 10.1145/2499370.2491973. Cited on page: 79

[44] John F. Couleur and Edward L. Glaser. Shared-access data processing system, US Patent 3,412,382. Cited on page: 2, 93

[45] Chen Ding and Ken Kennedy. Inter-array data regrouping. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 149–163, 2000. DOI: 10.1007/3-540-44905-1_10. Cited on page: 88

[46] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 229–237, 1999. Cited on page: 40

[47] Steve Dropsho, Alper Buyuktosunoglu, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, Greg Semeraro, Grigorios Magklis, and Michael L. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 141–, 2002. DOI: 10.1109/PACT.2002.1106013. Cited on page: 8, 95, 107, 108, 112

[48] Yu Du, Miao Zhou, B.R. Childers, D. Mosse, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, HPCA '15, pages 223–234, DOI: 10.1109/HPCA.2015.7056035. Cited on page: 90

[49] Dongrui Fan, Zhimin Tang, Hailin Huang, and Guang R. Gao. An Energy Efficient TLB Design Methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 351–356, 2005. DOI: 10.1145/1077603.1077688. Cited on page: 2, 4, 8, 21, 93, 94, 125

[50] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 37–48, 2012. DOI: 10.1145/2150976.2150982. Cited on page: 6, 31, 32, 33, 39, 47, 52, 57, 58, 61

[51] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Comput. Archit. News*, 42(2): 20–23, DOI: 10.1145/2669594.2669599. Cited on page: 8, 63, 80, 93, 94, 95, 103, 112, 126

[52] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 178–189, 2014. DOI: 10.1109/MICRO.2014.37. Cited on page: 6, 62, 63, 65, 80, 81, 89

[53] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, Cited on page: 9

[54] James R. Goodman. Coherency for multiprocessor virtual address caches. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 72–81, 1987. DOI: 10.1145/36206.36186. Cited on page: 27

[55] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd 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. DOI: 10.1145/2150976.2150994. Cited on page: 90

[56] Per Hammarlund. 4th generation Intel Core processor, codenamed Haswell. In *Proceedings of Hot Chips Symposium*, Cited on page: 24, 93, 97, 111

[57] Heather Hanson, M. S. Hrishikesh, Vikas Agarwal, Stephen W. Keckler, and Doug Burger. Static energy reduction techniques for microprocessor caches. *IEEE Trans. VLSI Syst.*, 11(3):303–313, DOI: 10.1109/TVLSI.2003.812370. Cited on page: 124

[58] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 3rd edition, Cited on page: 26

[59] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, DOI: 10.1145/1186736.1186737. Cited on page: 81, 96, 116

[60] Intel Corporation. Introduction to the iAPX 432 Architecture, Cited on page: 28, 90

[61] Intel Corporation. TLBs, Paging-Structure Caches and their Invalidation, Cited on page: 89, 97, 112

[62] Intel Corporation. *Intel® Processor Identification and the CPUID Instruction*. Number 241618-039 in Application Note 485, Cited on page: 34

[63] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-026, Cited on page: 36, 49, 50, 57, 71, 89, 97, 116

[64] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-046US, Cited on page: 17

[65] B. Jacob and T. Mudge. Software-Managed Address Translation. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA '97, pages 156–, 1997. DOI: 10.1109/HPCA.1997.569652. Cited on page: 27

[66] Bruce Jacob and Trevor Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, 18(4):60–75, DOI: 10.1109/40.710872. Cited on page: 11, 15, 16, 20, 28, 89, 90, 94, 126

[67] Bruce Jacob and Trevor Mudge. Virtual memory: Issues of implementation. *Computer*, 31(6):33–43, DOI: 10.1109/2.683005. Cited on page: 11, 15, 16, 20, 28

[68] Bruce L. Jacob and Trevor N. Mudge. A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 295–306, 1998. DOI: 10.1145/291069.291065. Cited on page: 2, 4, 21, 59

[69] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 404–415, 2013. DOI: 10.1145/2485922.2485957. Cited on page: 6, 32

[70] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing Data Analysis Workloads in Data Centers. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC 2013, pages 66–76, DOI: 10.1109/IISWC.2013.6704671. Cited on page: 34

[71] Toni Juan, Tomas Lang, and Juan J. Navarro. Reducing TLB Power Requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 196–201, 1997. DOI: 10.1145/263272.263332. Cited on page: 2, 4, 8, 21, 93, 94, 124

[72] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating Physical Addresses Directly for Saving Instruction TLB Energy. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO-35, pages 185–196, 2002. DOI: 10.1109/MICRO.2002.1176249. Cited on page: 2, 4, 21, 93, 94, 126

[73] Ismail Kadayif, Partho Nath, Mahmut T. Kandemir, and Anand Sivasubramaniam. Compiler-directed physical address generation for reducing dTLB power. In *ISPASS*, pages 161–168, 2004. DOI: 10.1109/ISPASS.2004.1291368. Cited on page: 2, 4, 21, 93, 94, 126

[74] Mahmut Kandemir, Ismail Kadayif, and Guilin Chen. Compiler-directed code restructuring for reducing data tlb energy. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*,

CODES+ISSS ’04, pages 98–103, 2004. DOI: 10.1145/1016720.1016747. Cited on page: 94, 126

[75] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-tlb behavior of spec cpu2000 benchmarks. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’02, pages 129–139, 2002. DOI: 10.1145/511334.511351. Cited on page: 38, 59

[76] Gokul B. Kandiraju and Anand 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. Cited on page: 58, 89

[77] Vasileios Karakostas, Osman S. Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance Analysis of the Memory Management Unit under Scale-out Workloads. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization*, IISWC 2014, pages 1–12, DOI: 10.1109/IISWC.2014.6983034. Cited on page: 2, 4, 9, 21, 61, 65, 89, 90, 94, 97, 116, 117

[78] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA ’15, pages 66–78, 2015. DOI: 10.1145/2749469.2749471. Cited on page: 8, 9, 93, 94, 95, 99, 100, 103, 110, 126

[79] Vasileios Karakostas, Jayneel Gandhi, , Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Energy-Efficient Address Translation. In *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture*, Cited on page: 9

[80] Stefanos Kaxiras and Alberto Ros. A new perspective for efficient virtual-cache coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA ’13, pages 535–546, 2013. DOI: 10.1145/2485922.2485968. Cited on page: 27, 53

[81] Joo-Young Kim and Hoi-Jun Yoo. Bitwise Competition Logic for Compact Digital Comparator. In *Proceedings of the 2007 IEEE Asian Solid-State Circuits Conference*, ASSCC, pages 59–62, DOI: http://dx.doi.org/10.1109/ASSCC.2007.4425682. Cited on page: 71

[82] Hsien-Hsin S. Lee and Chinnakrishnan S. Ballapuram. Energy Efficient D-TLB and Data Cache Using Semantic-aware Multilateral Partitioning. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ISLPED '03, pages 306–311, 2003. DOI: 10.1145/871506.871583. Cited on page: 8, 94, 125

[83] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 694–701, 2011. Cited on page: 95, 99, 113

[84] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *ISCA*, pages 36–47, DOI: 10.1145/2485922.2485926. Cited on page: 39

[85] William Lonehgan and Paul King. Design of the b 5000 system. *Datamation*, 7(5), Cited on page: 28, 90

[86] Pejman Lotfi-Kamran, Boris Grot, and Babak Falsafi. Noc-out: Microarchitecting a scale-out processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 177–187, 2012. DOI: 10.1109/MICRO.2012.25. Cited on page: 6, 32

[87] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-out Processors. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 500–511, 2012. DOI: 10.1145/2366231.2337217. Cited on page: 6, 32

[88] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings*

of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pages 190–200, 2005. DOI: 10.1145/1065010.1065034. Cited on page: 95, 99, 112

[89] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.*, 10(1):2:1–2:38, DOI: 10.1145/2445572.2445574. Cited on page: 45, 58, 90

[90] Collin McCurdy, Alan L. Coxa, and Jeffrey Vetter. Investigating the tlb behavior of high-end scientific applications on commodity microprocessors. In *Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '08, pages 95–104, 2008. DOI: 10.1109/ISPASS.2008.4510742. Cited on page: 2, 4, 21, 59

[91] MIPS Technologies, Incorporated. MIPS32 Architecture for Programmers Volume iii: The MIPS Privileged Resource Architecture, Cited on page: 23, 65, 97

[92] Alessandro Morari, Roberto Gioiosa, Robert W. Wisniewski, Bryan Rosenburg, Todd Inglett, and Mateo Valero. Evaluating the impact of TLB misses on future HPC systems. In *The 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2012)*, IPDPS 2012, pages 1010–1021, May 2012. DOI: http://dx.doi.org/10.1109/IPDPS.2012.94. Cited on page: 59

[93] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed tlbs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 27–38, 1993. DOI: 10.1145/165123.165127. Cited on page: 59

[94] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, pages 89–104, 2002. DOI: 10.1145/1060289.1060299. Cited on page: 65, 89

[95] Misel-Myrto Papadopoulou, Xin Tong, Andre Seznec, and Andreas Moshovos. Prediction-based superpage-friendly TLB designs. In *Proceedings of the 21st IEEE*

International Symposium on High Performance Computer Architecture, pages 210–222, DOI: http://dx.doi.org/10.1109/HPCA.2015.7056034. Cited on page: 24, 90, 94, 97, 114, 119, 124

[96] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 258–269, 2012. DOI: 10.1109/MICRO.2012.32. Cited on page: 6, 8, 58, 61, 64, 65, 76, 79, 89, 94, 95, 103, 126

[97] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing TLB reach by exploiting clustering in page translations. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*, pages 558–567, 2014. DOI: http://dx.doi.org/10.1109/HPCA.2014.6835964. Cited on page: 6, 8, 58, 61, 64, 65, 76, 82, 84, 89, 94, 95, 103, 126

[98] Stephen Phillips. M7: Next Generation SPARC. In *Hot Chips: A Symposium on High Performance Chips*, Cited on page: 88

[99] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, ISLPED '00, pages 90–95, 2000. DOI: 10.1145/344166.344526. Cited on page: 124

[100] Xiaogang Qiu and Michel Dubois. Towards Virtually-Addressed Memory Hierarchies. In *Proceedings of the 7th IEEE International Symposium on High Performance Computer Architecture*, pages 51–62, DOI: http://dx.doi.org/10.1109/HPCA.2001.903251. Cited on page: 27, 53

[101] Dino Quintero, Sebastien Chabrolles, Chi Hui Chen, Murali Dhandapani, Talor Holloway, Chandrakant Jadhav, Sae Kee Kim, Sijo Kurian, Bharath Raj, Ronan Resende, Bjorn Roden, Niranjan Srinivasan, Richard Wale, William Zanatta, and Zhi Zhang. IBM Power Systems Performance Guide Implementing and Optimizing, Cited on page: 23, 65, 97

[102] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-39, pages 423–432, 2006. DOI: 10.1109/MICRO.2006.49. Cited on page: 8, 95, 107, 108

[103] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 285–298, 1995. DOI: 10.1145/224056.224078. Cited on page: 59

[104] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB Preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 117–127, 2000. DOI: 10.1145/339647.339666. Cited on page: 58, 89

[105] Azam Seyedi, Vasileios Karakostas, Stefan Cosemans, Adrián Cristal, Mario Nemirovsky, and Osman S. Unsal. NEMsCAM: A novel CAM cell based on nano-electro-mechanical switch and CMOS for energy efficient TLBs. In *Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures*, pages 51–56, DOI: http://dx.doi.org/10.1109/NANOARCH.2015.7180586. Cited on page: 125

[106] André Seznec. A Case for Two-way Skewed-associative Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 169–178, 1993. DOI: 10.1145/165123.165152. Cited on page: 88

[107] Manish Shah, Robert Golla, Gregory Grohoski, Paul Jordan, Jama Barreh, Jeffrey Brooks, Mark Greenberg, Gideon Levinsky, Mark Luttrell, Christopher Olson, Zeid Samoail, Matt Smittle, and Thomas Ziaja. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *IEEE Micro*, 32(2):8–19, DOI: 10.1109/MM.2012.1. Cited on page: 23, 24, 65, 93, 97, 111

[108] Avinash Sodani. Race to Exascale: Opportunities and Challenges. In *MICRO Keynote*, Cited on page: 4, 93

[109] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *Proceedings of the 43rd Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO '43, pages 313–324, 2010. DOI: 10.1109/MICRO.2010.26. Cited on page: 90

[110] Sun Microsystems. UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007. Cited on page: 89

[111] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 171–182, 1994. DOI: 10.1145/195473.195531. Cited on page: 6, 8, 58, 61, 64, 65, 89, 94, 95, 103, 126

[112] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Trade-offs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 415–424, 1992. DOI: 10.1145/139669.140406. Cited on page: 24, 97

[113] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy 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. DOI: 10.1109/MICRO.2008.4771782. Cited on page: 71, 90

[114] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 140–148, 1989. DOI: 10.1145/74925.74942. Cited on page: 53

[115] Emmett Witchel, Josh Cates, and Krste 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. DOI: 10.1145/605397.605429. Cited on page: 90

[116] David A. Wood, Susan J. Eggers, Garth Gibson, Mark D. Hill, and Joan 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. DOI: 10.1145/17356.17398. Cited on page: 27, 53, 90, 94, 126

[117] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 2–11, 2011. DOI: 10.1109/ISPASS.2011.5762710. Cited on page: 57

[118] Wei Xu, Hongbin Sun, Xiaobin Wang, Yiran Chen, and Tong Zhang. Design of Last-level On-chip Cache Using Spin-torque Transfer RAM (STT RAM). *IEEE Trans. Very Large Scale Integr. Syst.*, 19(3):483–493, DOI: http://dx.doi.org/10.1109/TVLSI.2009.2035509. Cited on page: 58

[119] Jiachen Xue and Mithuna Thottethodi. PreTrans: Reducing TLB CAM-search via Page Number Prediction and Speculative Pre-translation. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 341–346, 2013. DOI: http://dx.doi.org/10.1109/ISLPED.2013.6629320. Cited on page: 125

[120] Hongil Yoon and Gurindar S. Sohi. Revisiting Virtual L1 Caches: A Practical Design Using Dynamic Synonym Remapping. In *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture*, Cited on page: 27