Efficient Memory Virtualization

by

Jayneel Gandhi

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 19th August 2016

To my parents Mayuri and Nalin Gandhi, my other half Shailee Thaker,
and my pet dog Pabu along with my family and friends
for their unconditional love and support.

ACKNOWLEDGMENTS

This thesis marks the end of my Ph.D. journey. It has been a long arduous one to say the least. It represents an important milestone in my life within Multifacet Group at University of Wisconsin. There are numerous people whose support, knowledge and encouragement has kept this thesis on track and seen it to completion. I would like to acknowledge these remarkable individuals in making this journey an unforgettable experience

First and foremost, I am indebted to my advisors Prof. Mark Hill and Prof. Michael Swift who guided me all the way through my graduate school. Their combination as a senior faculty looking for a broad high-level idea and a relatively newer faculty with the zeal to find technical depth, provided the balance needed to reflect on new ideas. Their expertise, in Computer Architecture and Operating Systems respectively, offered me with a seamless platform to do cross-layer research. Numerous discussion with them enabled me to evaluate ideas quickly for their efficacy and impact. Their attention to detail and insistence on pushing the envelope while developing any proposal and writing papers have molded me into the researcher that I am today.

Along with my advisors, my thesis committee steered me through these years and I am grateful for their valuable feedback. Thank you Kathryn McKinley, Mikko Lipasti, David Wood and Eftychios Sifakis. Kathryn is my role model for understanding how to collaborate with multiple people and bringing out the best in them. I cannot thank David enough for assisting me during a difficult time of my graduate school and helping me get a PhD advisor.

Working on a doctorate degree always involves engaging with a large diverse group, understanding different viewpoints and coming to an agreement after several exchanges. I take this opportunity to sincerely acknowledge my collaborators for making me realize the importance of being a team player. Particularly, I want to mention Arkaprava Basu who motivated me work with both Mark and Mike and directed me towards my dissertation research. His advice and contributions helped refine a lot of questions regarding my research. Another special acknowledgement, my collaborator Vasileios Karakostas on multiple projects some of which are also part of this dissertation. It was this association with him that made a lot of proposals come to fruition. I would also like to mention other notable collaborators: Prof. Osman Ünsal, Prof. Mario Nemirovsky, Dr. Adrián Cristal and the FabScalar Team at North Carolina State University.

I graciously thank the faculty members at University of Wisconsin-Madison, Prof. Karu Sankaralingam and Prof. Guri Sohi, and Wisconsin Computer Architecture Affiliates for providing timely feedback. I cannot forget Prof. Remzi Arpaci-Dusseau who helped build my foundations in Operating Systems and through course CS838: Virtualization, introducing an important area which formed an integral part of my thesis. My heartfelt thanks to Richardson Addai-Mununkum and Samuel Peters for proof-reading various drafts of my papers. I would also like to mention various people from the computer architecture community who has not only given feedback on my work but also very important career advice too. Specifically: Tom Wenish, Jason Mars, Lingjia Tang, Doug Burger, Babak Falsafi, Eric Rotenberg, Yan Solihin, Benjamin Serebrin, Mike Marty and Phillip Wells. I would also like to thank my colleagues at University of Wisconsin-Madison, in particular Lena Olson, Jason Lowe-Power, Rathijit Sen, Tony Nowatzki, Newsha Ardalani, Muhammad Shoaib Bin Altaf, Sanketh Nalli, Venkatanathan Varadarajan, Joel Hestness, Sankaralingam Panneerselvam, Nilay Vaish, Swapnil Haria, Somayeh Sardashti, Jaikrishnan Menon, Hongil Yoon, Gagan Gupta, Srinath Sridhar, Sujith Surendran, Dibakar Gope, Vignyan Reddy, Chris Feilbach, Aditya Venkataraman, Marc Orr, Dan Gibson, Derek Hower, Marc de Kruijf, and Venkatraman Govindaraju, for the many stimulating and inspiring dialogues involving technical and non-technical topics which shaped my perspective within and outside the academic field.

This work has been supported in part by the National Science Foundation (CNS-0720565, CNS-0834473, CNS-0916725, CNS-1117280, CCF-1218323, CNS-1302260, CCF-1438992, CCF-1533885), the European Union (FEDER funds) under contract TIN2012-34557, the European Union's Seventh Framework Programme (FP7/2007- 2013) under the ParaDIME project (GA no. 318693), Google, and the University of Wisconsin (Kellett award and Named professorship to Hill). This thesis might not have seen the light of day if not for these generous donations.

It would be remiss to not mention my friends who helped make this long journey incredibly enjoyable: Amritava Das, Kushal Sinha, Vikram Adhikarla, Sonalee Burthakur, Prachi Arora, Suarabh Ghanekar, Aditya Ghule, Krutu Garg, Kartik Nagendra, Vaishnavi Tripuraneni, Amulya Rao, Zachary Marzec, Hangjian Zhao, Brandon Smith, Laurie Stephey, Karthik Omanakuttan, Prabhdeep Virk, Kinjal Bhavsar, Anirudh Acharya, Vasant Agrawal, Varun Murthy, Akshay Pavagada, Hiren Patel, Hardik Parekh, Ruturaj Patel, Ajay Mulchandani. I want to pay homage to a friend and collaborator: Late Hiran Mayukh. We both had worked on FabScalar project at North Carolina State University and moved to the Ph.D. program at University of Wisconsin-Madison to fulfill our dreams. I will always miss our conversations on various tradeoffs that computer architecture exposes us to.

Last but not least, all of this would not have been possible without tremendous support, encouragement and unconditional love from my family. I profoundly thank my parents, Mayuri and Nalin Gandhi for supporting me always and encouraging me to follow my dreams. A big thank-you to Shailee Thaker, my fiancée, for providing her love and support through graduate school and ensuring my focus during this last mile of my doctorate marathon. A small but significant mention, my pet dog Pabu who has been an important part of my life at Madison. My extended family deserves acknowledgement for providing their unyielding support and love.

Besides this, several people have knowingly and unknowingly helped me in the successful completion of this project. My deepest and sincerest gratitude to all those and many more for helping me complete this journey on such a high note.

CONTENTS

Contents vi

List of Tables ix

List of Figures x

Abstract xii

1 Introduction 1
- *1.1 Virtual Memory and Paging* 2
- *1.2 Motivation* 4
  - *1.2.1 Growing Overheads of Paging* 4
  - *1.2.2 Meteoric Rise of Virtual Machines* 6
- *1.3 Proposals* 7
  - *1.3.1 Virtualized Direct Segments* 8
  - *1.3.2 Agile Paging* 10
  - *1.3.3 Redundant Memory Mappings* 11
- *1.4 Thesis Organization* 12

2 Background 14
- *2.1 Page-Based Virtual Memory* 14
  - *2.1.1 Page Table* 14
  - *2.1.2 Translation Lookaside Buffer* 15
  - *2.1.3 Native Hardware Page Table Walk* 16
  - *2.1.4 Page Walk Caches* 17
  - *2.1.5 Hierarchical TLBs* 19
  - *2.1.6 Huge Pages* 19
- *2.2 Support for Virtual Machines* 20
  - *2.2.1 Hardware-based Nested Paging* 21
  - *2.2.2 Hypervisor-enabled Shadow Paging* 23
  - *2.2.3 Page Walk Caches and Nested TLBs* 24
  - *2.2.4 Dimensionality of Page Walks* 25
- *2.3 Other Related Work* 25
  - *2.3.1 Virtual Memory* 26
  - *2.3.2 Multipage Mappings* 27
  - *2.3.3 Direct Segments* 28
  - *2.3.4 Virtualization* 29
  - *2.3.5 Flat Page Tables for Virtual Machines* 30
  - *2.3.6 Selective Hardware Software Paging (SHSP)* 30

2.3.7 Segmentation 30

3 BadgerTrap: A Tool to instrument x86-64 TLB misses **32**

3.1 Introduction 32

3.2 Design 34

3.2.1 Intercepting TLB misses 34

3.2.2 Software-assisted TLB miss handler 35

3.2.3 Enabling BadgerTrap 37

3.2.4 Instrumenting TLB misses 37

3.3 Using BadgerTrap 38

3.3.1 Study 1: Direct Segment 38

3.3.2 Study 2: Coalesced and Shared MMU 39

3.3.3 Study 3-5: This Thesis 39

3.3.4 Performance 40

3.3.5 Discussion 40

3.4 Limitations and Future Work for the Tool 41

4 Virtualized Direct Segments **43**

4.1 Introduction 43

4.2 Hardware Design and Software Support 48

4.2.1 Dual Direct mode 48

4.2.2 VMM Direct mode 50

4.2.3 Guest Direct mode 52

4.2.4 Unvirtualized Direct Segments 53

4.3 Reducing Memory Fragmentation 54

4.3.1 Self-ballooning 54

4.3.2 Reclaiming I/O gap memory 55

4.3.3 Memory compaction 56

4.4 Escape Filter 57

4.5 Prototype Implementation 58

4.5.1 Contiguous Allocation 58

4.5.2 Emulating Segments 59

4.5.3 Self-Ballooning Prototype 59

4.6 Evaluation Methodology 61

4.7 Cost of Virtualization 64

4.8 Evaluation of New Design 68

4.8.1 Performance Analysis 69

4.8.2 Energy Discussion 70

4.8.3 Escape Filter 72

4.8.4 Shadow Paging: An Alternative 73

4.8.5 Content-Based Page Sharing 74

4.9 Summary 75

5 Agile Paging **76**

5.1 Introduction 76

5.2 *Agile Paging Design* 81
5.2.1 *Mechanism: Hardware Support* 83
5.2.2 *Mechanism: VMM Support* 85
5.2.3 *Policies: What degree of nesting to use?* 88
5.3 *Hardware Optimizations* 91
5.4 *Paging Benefits* 92
5.5 *Methodology* 94
5.6 *Results* 98
5.6.1 *Performance analysis* 98
5.6.2 *Insights into Performance of Agile Paging* 101
5.6.3 *Discussion: Selective Hardware Software Paging (SHSP)* 101
5.7 *Summary* 102

6 Exploiting Range Translations with Redundant Memory Mappings 103
6.1 *Introduction* 103
6.2 *Redundant Memory Mappings* 109
6.2.1 *Overview* 110
6.3 *Architectural Support* 114
6.3.1 *Range TLB* 114
6.3.2 *Range table* 116
6.3.3 *Handling misses in the range TLB* 117
6.4 *Operating System Support* 119
6.4.1 *Managing range translations* 119
6.4.2 *Contiguous memory allocation* 120
6.5 *Discussion* 123
6.6 *Methodology* 125
6.7 *Results* 129
6.7.1 *Performance analysis* 131
6.7.2 *Range TLB sensitivity analysis* 132
6.7.3 *Impact of eager paging* 133
6.7.4 *Energy* 136
6.8 *Summary* 137

7 Summary and Future Work 138
7.1 *Summary* 138
7.2 *Future Work* 141

Bibliography 145

LIST OF TABLES

1.1 TLB sizes in Intel processors for last 15 years. 4

2.1 Details of the private, per-core, data TLB hierarchy for the three latest Intel processor architectures. 20

4.1 Steps in address translation of a guest virtual address in Dual Direct mode. . . 49
4.2 Trade-offs in various virtualized designs. 54
4.3 Various modes utilized in fragmented systems. 57
4.4 Workload discriptions. 62
4.5 Description of native and virtualized systems. 62
4.6 Linear model for cycle spent on page walk. 63

5.1 Trade-off provided by both memory virtualization techniques as compared to base native. Agile paging exceeds best of both worlds. 77
5.2 Number of memory references with varying degree of nesting provided by agile paging in a four-level x86-64-style page table as compared to other techniques. . 79
5.3 System configurations and per-core TLB hierarchy. 94
5.4 Performance model based on performance counters and BadgerTrap. 95
5.5 Workload description and memory footprint. 96
5.6 Percentage of TLB misses covered by each mode of agile paging while using 4KB pages assuming no page walk caches. Most of the TLB misses are served in shadow mode. 101

6.1 Comparison of Redundant Memory Mappings with previous approaches for reducing virtual memory overhead. 104
6.2 Total translation entries mapping the application's memory with: (i) Transparent Huge Pages of 4 KB and 2 MB pages [15] 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. 108
6.3 Overview of Redundant Memory Mapping 112
6.4 Workload description and memory footprint. 126
6.5 System configurations and per-core TLB hierarchy. 127
6.6 Performance model based on hardware performance counters and BadgerTrap. 129
6.7 Impact of eager paging on ranges, time, and memory compared to demand paging (with Transparent Huge Pages). 134

LIST OF FIGURES

1.1 Virtual to physical mappings using pages for two processes and its use by hardware. 3
1.2 Physical memory sizes purchased with $10,000 for the last 35 years. 4

2.1 Page table format for x86-64. 15
2.2 A TLB translating a virtual address to physical address. 16
2.3 Native address translation supported by hardware. 17
2.4 Pseudocode for native hardware page walk. 17
2.5 Organization of page walk caches in Intel processors. 18
2.6 Memory mapped by one entry with various proposals: (a) hierarchical translation look-aside buffers, (b) multipage mappings, (c) huge pages, and (d) direct segments. Each proposal tries to increase the reach of the TLB.. 19
2.7 Two-level address translation with virtual machines. 21
2.8 Nested page walk supported by hardware. 22
2.9 Pseudocode for hardware-based nested page walk state machine. The helper function host_walk is defined in Figure 2.4 22
2.10 Shadow page walk using the same hardware was the native page walk. 23
2.11 Pseudocode for shadow page walk uses host_walk as defined in Figure 2.4 . . . 23
2.12 Address space layout using direct segments. 29

3.1 Format of a 64-bit Page Table Entry [44]. 34
3.2 Flowchart for each translation with and without BadgerTrap along with state of PTE and TLB during an instrumented TLB miss. 35

4.1 Overheads associated with virtual memory for selected workloads and configurations. 44
4.2 Native and virtualized address translation modes supported by proposed hardware. 45
4.3 (a) Address translation flow chart (b) Steps of page walk state machine for various supported modes. 47
4.4 Memory layout for Dual Direct mode. 49
4.5 Memory layout for VMM Direct mode. 50
4.6 Memory layout for Guest Direct mode. 52
4.7 Illustration of guest physical memory with self-ballooning. 55
4.8 KVM memory slots to nested page tables. 60
4.9 Virtual memory overhead for each configuration per big-memory workload. . . 66
4.10 Virtual memory overhead for each configuration per compute workload. 67
4.11 Normalized execution time for big-memory workloads in presence of bad pages. 72

5.1 Different techniques of virtualized address translation as compared to base native. Numbers indicate the memory accesses to various page table structures on a TLB miss in chronological order. The merge arrows denotes that the two page tables are merged to create shadow page table. Colored merge arrows with agile paging denotes partial merging at that level. The starting point for translating an address is marked in bold and red. 80
5.2 Different degrees of nesting with agile paging in increasing order of page walk latency. Starting point for each is marked bold and green. 82
5.3 Pseudocode of hardware page walk state machine for agile paging. Note that agile paging requires modest switching support (shown in red) in addition to the state machines of nested and shadow paging shown in Sections 2.2.1 & 2.2.2 respectively. 83
5.4 Execution time overheads due to page walks (bottom bar) and VMM interventions (top dashed bar) for all workloads. 99

6.1 Range translation: an efficient representation of contiguous virtual pages mapped to contiguous physical pages. 106
6.2 Cumulative distribution function of the application's memory (percentage) that N translation entries map with pages (solid) and with optimal ranges (dashed), for seven representative applications. Ranges map all applications' memory with one to four orders of magnitude fewer entries than pages. 109
6.3 Redundant Memory Mappings design. The application's memory space is represented *redundantly* by both pages and range translations. 111
6.4 RMM hardware support consists primarily of a range TLB that is accessed in parallel with the last-level page TLB. 113
6.5 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. 116
6.6 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*. 121
6.7 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. 130
6.8 Range TLB miss ratio as a function of the number of range TLB entries. 133

7.1 Effect of different proposals in this thesis on dimensionality of page walk. . . . 140

ABSTRACT

Two important trends in computing are evident. First, computing is becoming more data centric, where low-latency access to a very large amount of data is critical. Second, virtual machines are playing an increasing critical role in server consolidation, security and fault tolerance as substantial amounts of computing migrate to shared resources in cloud services. Since the software stack accesses data using virtual addresses, fast address translation is a prerequisite for efficient data-centric computation and for providing the benefits of virtualization to a wide range of applications. Unfortunately, the growth in physical memory sizes is exceeding the capabilities of the most widely used virtual memory abstraction—paging—that has worked for decades.

This thesis addresses the above challenge in a comprehensive manner proposing a hardware/software co-design for fast address translation in both virtualized and native systems to address the needs of a wide variety of big-memory workloads. This dissertation aims to achieve near-zero overheads for virtual memory for both native and virtualized systems.

First, we observe that the overheads of page-based virtual memory can increase drastically with virtual machines. We previously proposed direct segments, which use a form of contiguous allocation in memory along with paging to largely eliminate virtual memory overhead for big-memory workloads on unvirtualized hardware. However, direct segments

are limited because they require programmer intervention and only only one segment is active at once. Here we generalize direct segments and propose *Virtualized Direct Segments* hardware with three new virtualized modes that significantly improves virtualized address translation. The new hardware bypasses either or both levels of paging for most address translations using direct segments. This preserves properties of paging when necessary and provides fast translation by bypassing paging where unnecessary.

Second, we found that virtualized direct segments bypassed widely used hardware support—nested paging—but ignores a less often used, but still popular, software technique—shadow paging. We show shadow paging provides an opportunity to reduce TLB miss latency while retaining all the benefits of virtualized paging. Nested and shadow paging provide different tradeoffs while managing two-levels of translation. To this end, we propose *agile paging*, which combines both techniques while preserving all benefits of paging and achieves better performance. Moreover, the hardware and operating systems changes for agile paging are much more modest than virtualized direct segments making it more practical for near term adoption.

Third, we saw that direct segments traded the flexibility of paging for performance, which is good for some applications, but was insufficient for many big-memory workloads. So, inspired by direct segments, we propose range translations that exploit virtual memory contiguity in modern workloads and map any number of arbitrarily-sized virtual memory ranges to contiguous physical memory pages while retaining the flexibility of paging. A range translation reduces address translation to a range lookup that delivers near-zero virtual memory overhead.

This thesis provides novel and modest address translation mechanisms, that improves

performance by reducing cost of address translation. The resulting system delivers a virtual memory design that is high performance, robust, flexible and completely transparent to the applications.

Chapter 1

INTRODUCTION

There are two important trends in computing that are more evident. First, computing is becoming more data centric, where low-latency access to a very large amount of data is critical. This trend is largely supported by increasing role of big-data applications in our day-to-day lives. Second, virtual machines are playing a critical role in server consolidation, security and fault tolerance as substantial computing migrates to shared resources in cloud services. This trend is evident due to increasing support for public, enterprise and private cloud services by various companies.

These trends put a lot of pressure on the virtual memory system—a layer of abstraction designed for applications to manage physical memory easily. The virtual memory system translates virtual addresses issued by the application to physical addresses for accessing the stored data. Since the software stack accesses data using virtual addresses, fast address translation is a prerequisite for efficient data-centric computation and for providing the benefits of virtualization to a wide range of applications. But unfortunately, growth in physical memory sizes is exceeding the capabilities of the virtual memory abstraction—paging. Paging has been working well for decades in the old world of scarce physical memory, but falls far short in the new world of gigabyte-to-terabyte memory sizes.

In this chapter, we will first review the concept of page-based virtual memory, then motivate the problem with paging in the new world, and finally introduce this thesis's contributions to mitigate the problem.

1.1 Virtual Memory and Paging

Virtual memory is a crucial abstraction to use physical memory in today's computer systems. It delivers benefits such as security due to process isolation and improved programmer productivity due to simple linear addressing. Each process has a very large virtual address space managed at granularity of pages, typically 4KB in size. Each virtual page is mapped to a physical page by the operating system (OS). All per-process virtual-to-physical mappings are stored in a page table, which is managed by the OS. Most commercial processors today use page-based virtual memory.

With virtual memory, the processor must translate every load and store generated by a process from a virtual address to a physical address to access the data. Because address translation is on the processors' critical path, a hardware structure called Translation Lookaside Buffer (TLB) accelerates translation by caching the most recently used page table entries (PTEs). Paging delivers high performance when most translations are serviced by TLB hits. However, a TLB miss requires a long latency operation to fetch the virtual-to-physical mapping, which is the corresponding PTE from the per-process page table.

A TLB miss can be handled in two ways. First, a lesser used technique handles a TLB miss using OS support by causing a trap into the OS on a TLB miss. This TLB miss trap enables the OS to walk the per-process page table, get the corresponding PTE and insert it into the TLB. This TLB miss trap is a very long latency operation (100s of processor cycles). This technique allows the OS to use any data structure to store the page table and allows

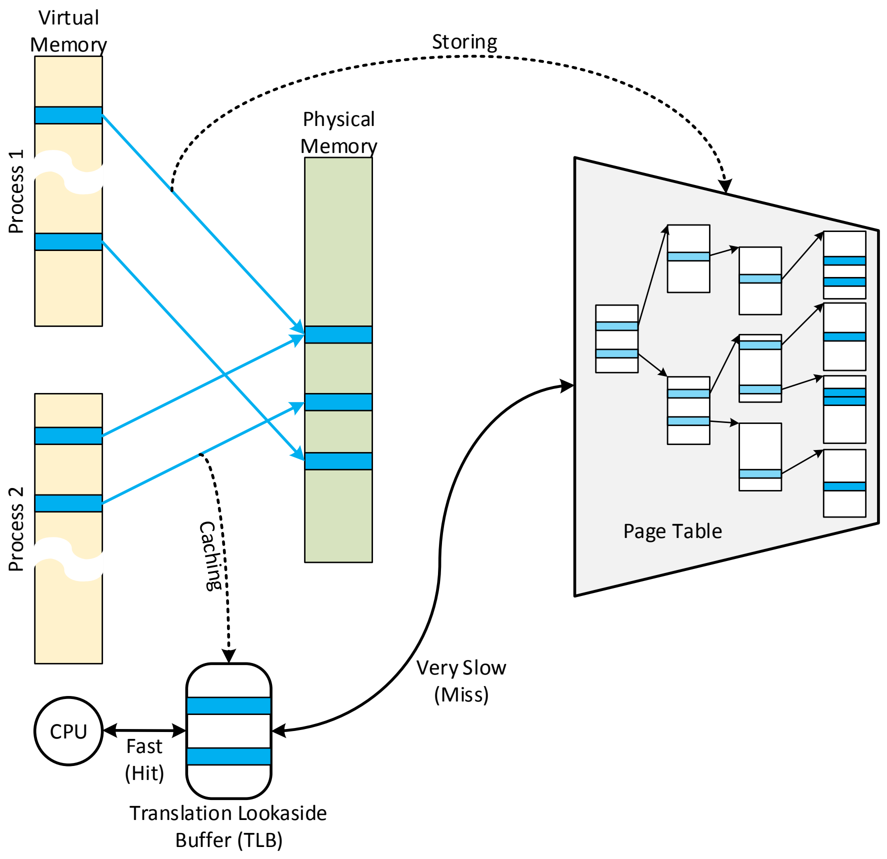

Figure 1.1: Virtual to physical mappings using pages for two processes and its use by hardware.

the OS to change the data structure of the page table at any time. This technique with some optimizations is used today in the SPARC architecture.

Most current architectures (including x86, ARM) uses the second widely used technique to handle TLB miss completely in hardware. On a TLB miss, the hardware triggers a state machine called the page table walker that walks the per-process page table and loads the corresponding PTE into the TLB. This is considered more efficient since this allows application to perform speculative execution, partially hide the latency of a TLB miss and does not require a trap. Since the hardware requires the knowledge about the data structure representing the page table, the page table is fixed for an architecture with a hardware page table walker. Figure 1.1 shows virtual memory for two processes being mapped to physical

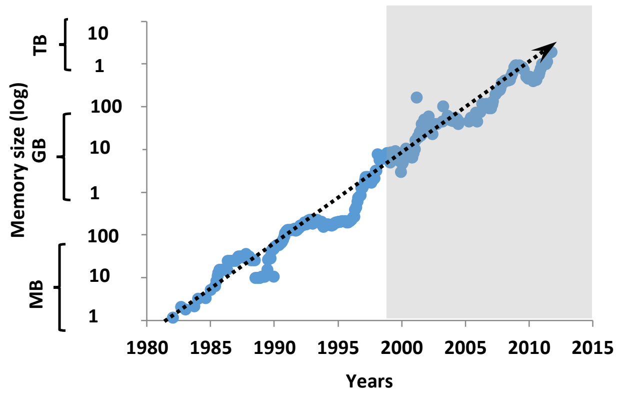

Figure 1.2: Physical memory sizes purchased with $10,000 for the last 35 years.

Year	Processor	L1 TLB size	L2 TLB size
1999	Pentium III	72	0
2004	Pentium 4	64	0
2008	Nehalem	96	512
2012	Ivybridge	100	512
2014	Haswell	100	1024
2015	Skylake	100	1552

Table 1.1: TLB sizes in Intel processors for last 15 years.

memory using pages. These mappings are cached in TLB for fast lookup. However, on a TLB miss, long latency operation is required to fetch the corresponding PTE whether handled by the OS or the hardware.

1.2 Motivation

There are two main factors that motivates us to rethink how we perform address translation: growing overheads of page-based virtual memory and the rise of virtual machines.

1.2.1 Growing Overheads of Paging

Unfortunately, large server workloads are experiencing execution time overheads of up to 50% due to paging on native systems [25, 30, 68, 69]. The following two opposing technology trends are at the root of this problem:

(i) Physical memory is growing exponentially cheaper and larger (Figure 1.2) allowing modern workloads to store ever increasing large data sets in memory. For example, Intel Haswell Xeon servers released in 2015 support up to 12 TB of memory with an 8-socket configuration, which are used for large databases and other workloads.

(ii) TLB sizes have grown slowly, because TLBs are on the processor's critical path to access memory (Table 1.1). The L1 TLB size has already stagnated, but even the size of L2 TLB is not growing as quickly as the memory demand.

The above trends causes a problem that is commonly called *limited TLB reach*—the fraction of physical memory that TLBs can map is reducing with each hardware generation. For instance, recently introduced Intel's Skylake processors released in 2015 has a TLB that can cover only 9% of a 256 GB memory. We expect this mismatch between TLB reach and memory size

(i) to keep growing as the physical memory available keeps on growing,

(ii) to become worse with newer memory technologies, which promise petabytes to zetabytes of physical memory, and

(iii) to increase the overheads of page-based address translation due to the increase in TLB misses.

Some previous approaches to address these overheads include hierarchical TLBs (adding larger but slower L2 TLBs), multipage mappings (mapping several pages with a single TLB entry), huge pages (mapping much larger aligned memory with a single TLB entry), and direct segments (providing a single arbitrarily large segment along with standard paging). These approaches are discussed in detail in Chapter 2. None of these approaches delivers a complete solution that solves the TLB reach problem while retaining flexible memory use.

These trends warrants rethinking about how we design virtual memory while preserving the benefits of paging.

1.2.2 Meteoric Rise of Virtual Machines

Virtual Machine Monitors (VMMs)—such as Xen [23], KVM [72], VMware [100]—provide an abstraction layer between OSes and the hardware. Virtualization has been used since the 1970s [56], but it's renaissance began with Disco in 1997 [39] and progressed without hardware support [17, 23] (e.g., dynamic binary translation [17, 20, 41, 104]) and, after gaining popularity, with hardware support [28, 29]. Virtualization enables better resource management, server consolidation, isolation, and fault tolerance—all very important in the era of cloud computing where virtual machines rule today. Virtualization in the cloud provides the added benefit of on-demand access to hardware and multi-tenancy to improve server utilization.

However, many of these benefits come with overheads in virtualizing processing, I/O, and memory. Fortunately, hardware advances in virtualization [28, 29, 63] (e.g., virtualizing I/O in hardware), have reduced many of these overheads substantially. But, overheads for virtualizing memory are still not universally low [17, 28, 38, 53]. The overheads come from TLB misses where hardware performs a nested two-level translation—first, from guest virtual address (gVA) to guest physical address (gPA); second from gPA to host physical address (hPA). This additional cost increases the latency of hardware page table walk and makes virtualization less attractive to memory-intensive applications. We observed that with limited TLB reach, the overheads of address translations can be 2-3x worse than that of native execution with memory-intensive applications [53]. This observation motivates low-cost mechanisms that will provide memory virtualization for free and make it attractive.

1.3 Proposals

In this thesis, I will address the challenge of high overheads of page-based virtual memory in a comprehensive manner to allow fast address translation in both the virtualized and native system for a wide variety of workloads. My dissertation research aims to achieve near-zero overheads of virtual memory for both native and virtualized systems primarily through three contributions.

First, we observed that the overheads of page-based virtual memory can increase drastically with virtual machines. Our earlier proposal of direct segments gave an opportunity to reduce these overheads. Direct segments uses a form of segmentation along with paging to largely eliminate virtual memory overhead for big-memory workloads on unvirtualized hardware (discussed in Section 2.3.3). We extend direct segments and propose *Virtualized Direct Segments* [53], new hardware building on direct segments [25] with three new virtualized modes that significantly improves virtualized address translation. The new hardware bypasses either or both levels of paging for most address translations using direct segments. This preserves properties of paging when necessary and provides fast translation by bypassing paging where unnecessary.

Second, we noticed that virtualized direct segments bypassed widely used hardware support—nested paging—but ignores a lesser used software technique—shadow paging.

Shadow Paging provides an opportunity to reduce TLB miss latency while retaining all the benefits of paging. Nested and shadow paging provide different tradeoffs while managing two-levels of translation. So, we propose *agile paging* [54], which combines both the techniques while preserving all benefits of paging and achieves better performance. Moreover, agile paging provides benefits with hardware and operating systems changes much more modest than virtualized direct segments.

Third, we observed that direct segments [25] traded the flexibility of paging for performance, which is good for some applications, but not all. So, I collaborated with fellow graduate student Vasileios Karakostas from Barcelona Supercomputing Center. Together we propose range translations that exploits virtual memory contiguity in modern workloads to perform address translation much more efficiently than paging while retaining all the benefits paging. I will briefly describe these three works below.

1.3.1 Virtualized Direct Segments

Virtualization provides value for many workloads, but its cost rises for workloads with poor memory access locality. We observed that there is significant overhead of address translation in the native system and the overheads increase drastically with virtualization. This overhead comes from TLB misses where the hardware performs a 2D page walk (up to 24 memory references on x86-64) rather than a native TLB miss (up to 4 memory references). This work is one of the first to study the memory overhead of virtualization for virtual machines with a large amount of memory.

To eliminate these overheads and make virtualization attractive, we propose *Virtualized*

Direct Segments [53], which extends our earlier proposal of direct segments [25] and support three different virtualized modes of operation to reduce address translation overheads in virtual machines. Each of these modes uses direct segments at either or both levels of address translations to bypass paging and reduce overheads of paging. At a given time, the system administrator can choose one of these modes to trade-off lower address translation overheads with higher execution environment flexibility and modifications to system software.

We further observed that a virtual machine may lack the contiguous guest physical memory needed to create a direct segment due to fragmentation. We address this problem with self-ballooning, which uses ballooning [100] and memory hotplug [8] to create contiguous physical guest memory from fragmented guest physical memory. Moreover, with hard faults in memory becoming increasingly prevalent [61], a handful of faulty pages can hinder the use of direct segment for large parts of memory. We thus propose an escape filter—a bloom-filter holding addresses of faulty pages—to enable use of direct segment hardware even in the presence of a few faulty pages.

Our results show that one mode—VMM Direct—reduces address translation overhead to near-native without guest application or OS changes (2% slower than native on average), while a more aggressive mode—Dual Direct—on big-memory workloads performs better-than-native with near-zero translation overhead.

1.3.2 Agile Paging

The previous proposal was used to bypass paging in widely used hardware support—nested paging—but ignored a lesser used software technique—shadow paging. Shadow paging provides an opportunity to reduce reduce TLB miss latency while retaining all the benefits of paging. Nested and shadow paging provide different tradeoffs while managing two-levels of translation. With nested paging, TLB misses are costly due to nested two-level page walk, but page table updates are fast. Whereas with shadow paging, TLB misses are short since hypervisor creates a single-level shadow page table, but requires costly hypervisor intervention on page table updates.

We observed that most of the higher levels of the the page table remains static and could benefit from low-latency page walk of shadow mode while dynamically changing lower levels can benefit from nested paging to lower the number of costly VMM intervention. Thus, we propose *agile paging* [54], which combines both the techniques and exceeds the best of both. A virtualized page walk starts in shadow mode and then optionally switches to nested mode for address ranges where frequent updates would cause costly hypervisor interventions. This reduces the cost of page walks, as many TLB misses can be handled partially or entirely in shadow mode while keeping VMM interventions required to a minimum. Agile paging builds on the existing techniques and only requires modest hardware support: the ability to switch between the two constituent techniques. In addition, we propose two optional hardware optimizations that further reduce VMM interventions for the shadow mode of agile paging. This technique preserves all benefits of paging and achieves better performance.

Our results show agile paging performs more than 12% better than both of its constituent techniques and comes within 4% of native execution for all workloads.

1.3.3 Redundant Memory Mappings

Our earlier proposal of direct segments [25] traded the flexibility of paging for performance, which is good for some applications, but not all. So, I collaborated with fellow graduate student Vasileios Karakostas from Barcelona Supercomputing Center to find a robust solution to reduce overheads of virtual memory in a native system. We observed that many applications naturally exhibit an abundance of contiguity in their virtual address space. If the OS can map this virtual contiguity to physical contiguity, a single entry is sufficient to translate from a virtual range to a physical range. By using the above observation, we introduce the key concept of range translation [55, 68] that exploits the virtual memory contiguity in modern workloads to perform address translation much more efficiently than paging. Inspired by direct segments [25], a range translation is a mapping between contiguous virtual pages mapped to contiguous physical pages of arbitrary size with uniform protection bits.

We implement range translations in the Redundant Memory Mappings (RMM) architecture. RMM employs hardware/software co-design to map the entire virtual address space with standard paging and redundantly map ranges with range translations. Since range translations are backed by page mappings in RMM, the operating system can flexibly choose between using range translations or not, retaining the benefits of paging for fine-grain memory management when necessary. The RMM system (i) efficiently caches range translations in a hardware range TLB to increase TLB reach, (ii) manages range translations using a per-process software range table just like the page table, and (iii) increases physical contiguity to increase the range size resulting in modest number of range translations per-process using eager paging.

Our results shows that RMM performs substantially better than paging alone and improves a wider variety of workloads than direct segments (one range per program), reducing the overhead of virtual memory to less than 1% on average.

1.4 Thesis Organization

Chapter 2 describes the state of the art of page-based virtual memory systems and its virtualization support. In addition, we will describe some recent other research proposals that we will use as comparisons for our proposals and range of related work

Chapter 3 explains the BadgerTrap tool [52], which we developed and have released online. This tool was originally published in Computer Architecture News (CAN), September 2014 issue. This tool will serve as the base for evaluating all the three proposals.

Chapter 4 describes Virtualized Direct Segments [53] that was originally published in the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47) 2014.

Chapter 5 describes Agile Paging [54] that was originally published in the 43rd Annual International Symposium on Computer Architecture (ISCA-43) 2016.

Chapter 6 describes Redundant Memory Mappings [68] that was originally published in the 42nd Annual International Symposium on Computer Architecture (ISCA-42) 2015.

A shorter version of this work is published in IEEE Micro Special Issue: Micro's Top Picks from Architecture Conferences 2016 [55].

Chapter 7 concludes the thesis and provides various potential extensions to this thesis.

Chapter 2

BACKGROUND

This chapter first presents state-of-the-art mechanisms and optimizations available today in our commodity processors that support page-based virtual memory, and more recent research proposals that help mitigate overheads associated with virtual memory. Most of the description follows x86-64 architecture, but these concepts are applicable to other architectures such as ARM and SPARC.

2.1 Page-Based Virtual Memory

The concept of virtual memory was first invented in 1959 with the Atlas system [70]. It was greatly refined with introduction of fixed size allocation in terms of *pages* in 1968 with the MULTICS systems [46]. Since that time, page-based virtual memory has dominated commercial computer architectures. We will next describe various components that makes this abstraction work efficiently in today's computer systems.

2.1.1 Page Table

The page table is a per-process structure which stores all virtual to physical mappings for each process. This structure is managed by the OS and walked by the hardware or OS to fetch a translation. For hardware based page walkers the structure of the page table is architecturally defined since hardware has access to the page table. A hierarchical page

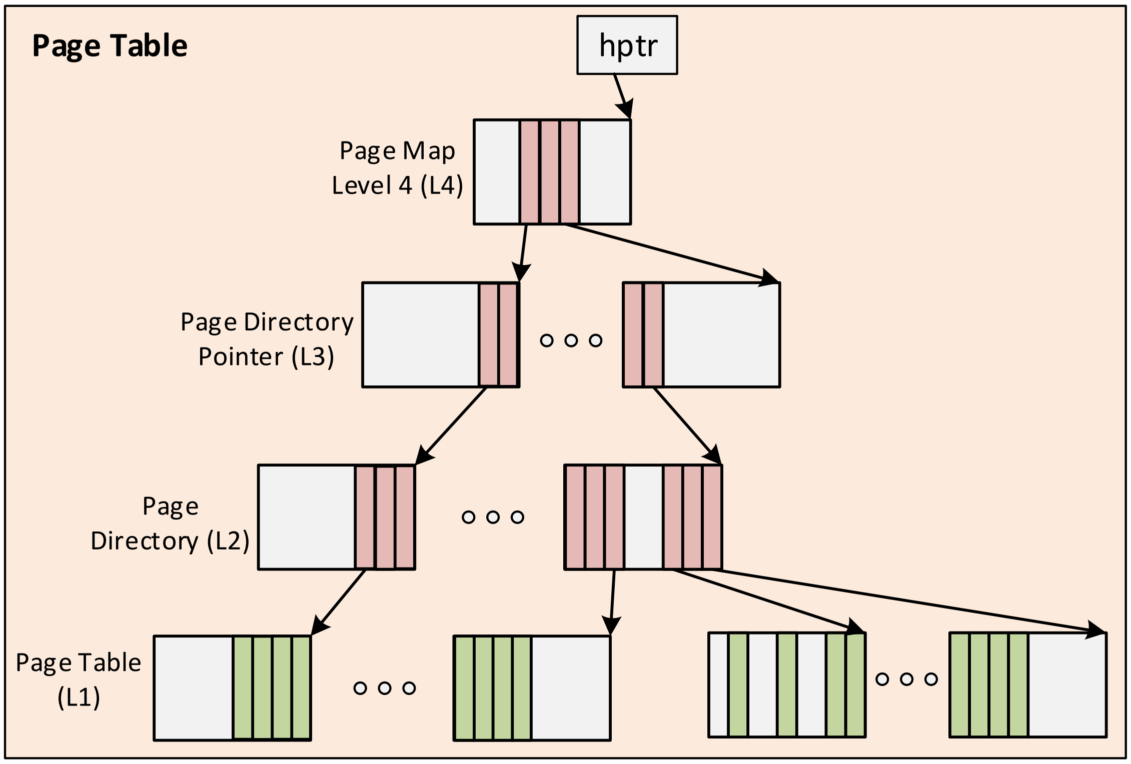

Figure 2.1: Page table format for x86-64.

table provides a compact space-efficient representation from a very large but sparsely populated virtual address space to smaller physical address space. For x86-64, a four-level hierarchical page table is used to store all virtual to physical mappings (see Figure 2.1). A hptr register stores the physical address that points to the top-level of the page table. Each page can store 512 entries and thus, each level of the page table can have a maximum fan out of 512.

2.1.2 Translation Lookaside Buffer

The most crucial hardware component that enables fast address translation is the *Translation Lookaside Buffer (TLB)* that caches most recently used page table entries (PTEs). A TLB is a hardware cache, which is indexed by part of a virtual address called virtual frame number.

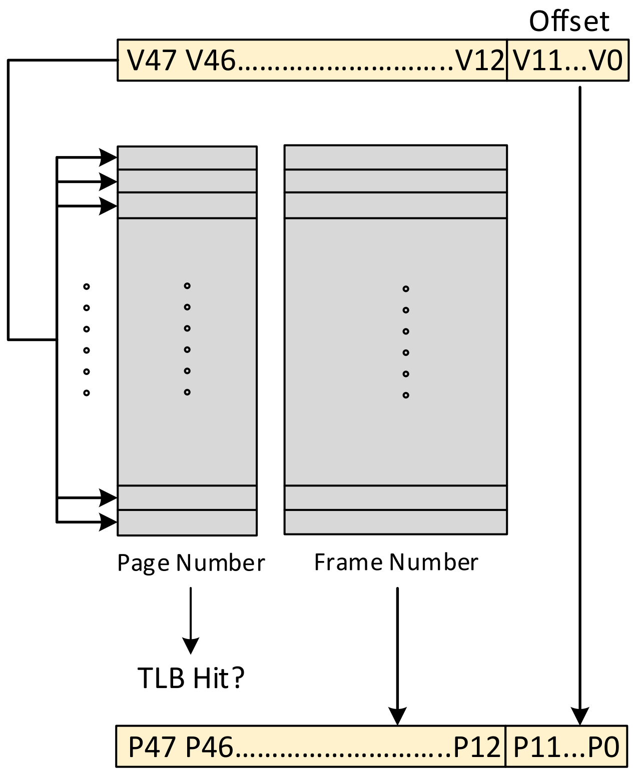
Offset
V47 V46..........................V12
V11...V0
Page Number
Frame Number
TLB Hit?
P47 P46..........................P12
P11...P0

Figure 2.2: A TLB translating a virtual address to physical address.

On a hit, it provides a physical frame number, which can be used to generate physical address along with the protection bits for that page. A TLB can be designed as fully associative, set-associative or direct-mapped cache. Figure 2.2 shows lookup operation in a TLB to translate a virtual address to physical address. The TLB lookup operation is performed for every load, store, and instruction fetch issued by a processor and mostly accesses in parallel with L1 cache lookup.

2.1.3 Native Hardware Page Table Walk

In case of a TLB miss, the corresponding PTE has to be fetched from the page table and introduced in the TLB. For hardware based page table walk in an architecture like x86-64, a hardware state machine is triggered with the virtual address to walk the page table. In

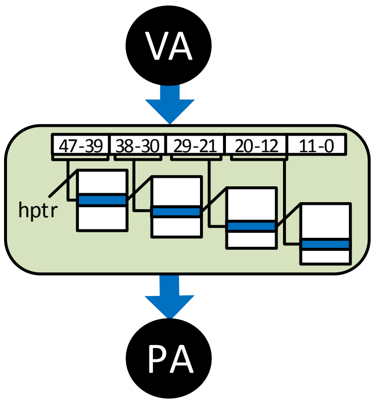

Figure 2.3: **Native address translation supported by hardware.**

```
host_walk(VA, hptr)

PA = hptr;
for (i=0; i⩽MAX_LEVELS; i++) do
    PA = host_PT_access(PA + index(VA,i));
end
return PA;

host_PT_access(address)

PTE = *address;
PA = PTE.PA;
if PTE.valid == 0 then
    raise host PAGE_FAULT;
return PA;
```

Figure 2.4: Pseudocode for native hardware page walk.

x86-64, there is a per-core hardware register called *cr3*, which stores the physical address that points to the top-level of the corresponding page table and its value is the same for a given process. For generality, we will call this pointer as *host pointer or hptr* interchangeably in this thesis. Using hptr in combination with virtual address, the four-level hierarchical page table is walked to retrieve the corresponding PTE. Figure 2.3 pictorially shows the native page table walk with which part of virtual address is used to create index at each level of page walk. Figure 2.4 describes the pseudocode for hardware page walker used to perform the native page walk. This page walk is also referred as 1D page walk. Note that this page walk will require 4 memory accesses to get the PTE to perform the translation.

2.1.4 Page Walk Caches

To reduce the latency of the 1D page walk from 4 memory accesses, page walk caches (PWCs) were invented [24, 30]. PWCs stores most recent partial translations which help reduce the latency of 1D page walk in best case to 1 memory access. The key observation is that the higher three levels of the page table shows high temporal locality and not the

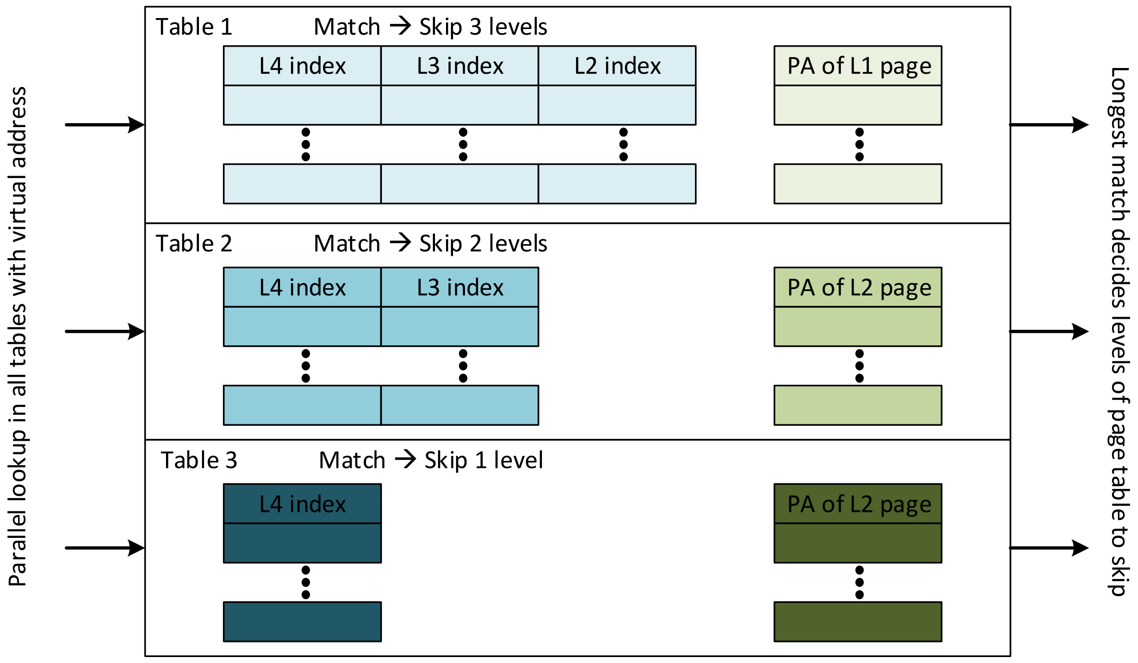

Figure 2.5: Organization of page walk caches in Intel processors.

last level. For example, page walks for two consecutive virtual pages would have the same three upper level entries since the higher index bits between both pages would be the same.

In this section, we will discuss Intel x86-64 style page walk cache and build on them in later chapters. We refer to each level of the x86-64 page table as L1 being the lowest level storing the PTEs and L4 being the highest level or the root. In Intel x86-64, page walk caches are designed as three tables. Each table stores L4 index or L4+L3 index or L4+L3+L2 index of a virtual page as tag and stores the physical address of the next level of the page table (see Figure 2.5). All three tables are looked up in parallel on a TLB miss with a virtual page number and there can be hits in any of the tables. But the longest hit is used to skip the maximum number of levels of the page table. Thus, in the best case, three levels of the page table walk are skipped bringing down the number of memory accesses from 4 to 1.

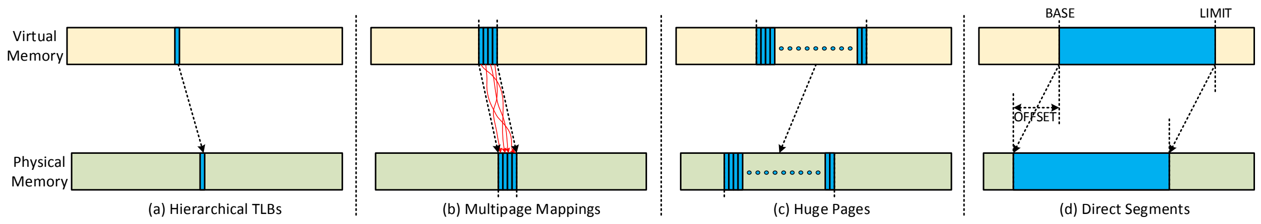

Figure 2.6: Memory mapped by one entry with various proposals: (a) hierarchical translation look-aside buffers, (b) multipage mappings, (c) huge pages, and (d) direct segments. Each proposal tries to increase the reach of the TLB..

2.1.5 Hierarchical TLBs

Two-level TLBs form a common organization for address translation in today's processors [58, 93]. The TLB organization is per-core. 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 [65]). To boost the performance further, processors provide separate TLBs for data and instructions. This technique helps increase the reach of the TLB by increasing the number of TLB entries, but does not increase the TLB reach of each TLB entry (see Figure 2.6(a)).

2.1.6 Huge Pages

Huge Pages using Transparent Huge Pages (THP) [15] and libhugetlbfs [1] increase the TLB reach and reduce the performance overhead of page walks [25, 30, 69, 79] by mapping a large fixed size region of memory with a single TLB entry [64, 76, 85, 93] (see Figure 2.6(c)). The x86-64 architecture supports mixing 4 KB with 2 MB and 1 GB pages at the same time. The hardware support for huge pages usually includes either a separate set associative L1 TLB for each page size, as in Intel processors [58], or a single fully associative L1 TLB that

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	n/a
2 MB	32	4-way	2 MB	—							
1 GB	4	fully	1 GB	—		1 GB	—		1 GB	16	n/a

Table 2.1: Details of the private, per-core, data TLB hierarchy for the three latest Intel processor architectures.

supports both 4 KB and huge pages, as in SPARC and AMD processors [18, 93]. 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 [79, 97]. 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 [81, 82]. In addition, many commodity processors provide limited numbers of large page TLB entries (see Table 2.1), which further limits their benefit [25, 53, 69].

To summarize, Table 2.1 overviews the details of the per-core TLB hierarchy for the recent Sandy Bridge and Haswell, and 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 that 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.

2.2 Support for Virtual Machines

To run multiple OSes on top of hardware, virtualization introduces a layer of indirection called the VMM or hypervisor. Similarly, to fully virtualize memory fully, a layer of

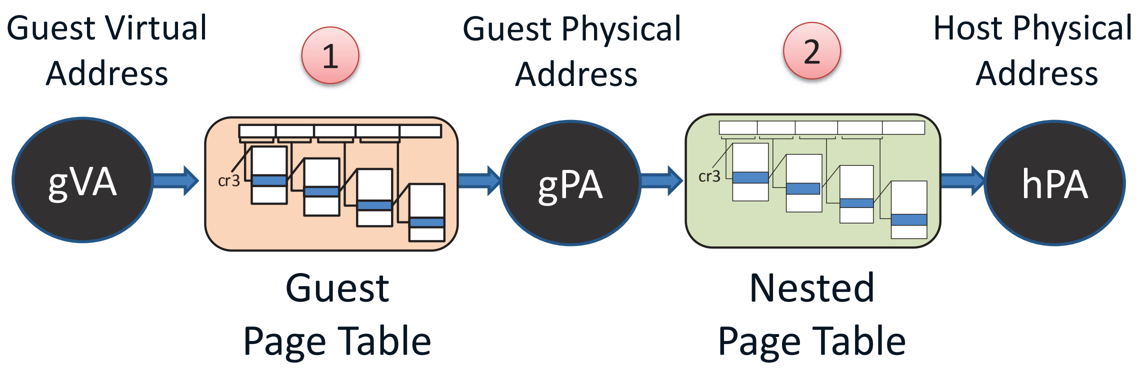

Figure 2.7: Two-level address translation with virtual machines.

indirection is introduced between guest virtual address (gVA) space and host physical address (hPA) space called the guest physical address (gPA) space. A two-level address translation is thus required with virtual machines (Figure 2.7):

gVA⇒gPA: guest virtual address to guest physical address translation via a per-process guest OS page table (gPT)

gP⇒hPA: guest physical address to host physical address via a per-VM host page table (hPT)

We will discuss two state-of-the-art techniques to manage and translate two-levels of page tables: nested paging and shadow paging.

2.2.1 Hardware-based Nested Paging

Nested paging is a widely used hardware technique to virtualize memory [28]. The processor has two hardware page table pointers to perform a complete translation: one points to the guest page table (gptr) and the other points to the host page table (hptr). The guest page table holds gVA to gPA translation and the host page table holds gPA to hPA translations.

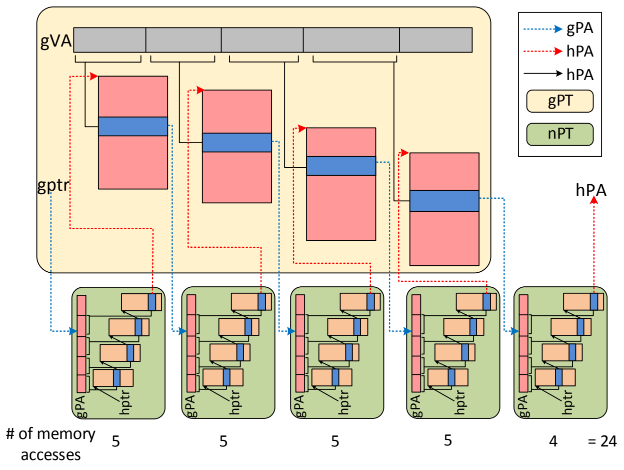

Figure 2.8: Nested page walk supported by hardware.

```
nested_walk(gVA, gptr, hptr)
//page fault in hPT will cause a VM exit
hPA = host_walk(gptr, hptr);
for (i=0; i≤MAX_LEVELS; i++) do
    hPA = nested_PT_access(hPA + index(gVA,i), hptr);
end
return hPA;

nested_PT_access(address, hptr)
PTE = *address;
gPA = PTE.PA;
if PTE.valid == 0 then
    raise guest PAGE_FAULT;
//page fault in hPT will cause a VM exit
hPA = host_walk(gPA, hptr);
return hPA;
```

Figure 2.9: Pseudocode for hardware-based nested page walk state machine. The helper function host_walk is defined in Figure 2.4

In the best case, the virtualized address translation hits in the TLB to directly translate from gVA to hPA with no overheads. In the worst case, a TLB miss needs to perform a 2D page walk that multiplies overheads vis-á-vis native, because accesses to the guest page table also require translation by the host page table. Figure 2.8 depicts virtualized address translation for x86-64. It shows how the page table memory references grow from a native 4 to a virtualized 24 references: 4 access to translate gptr (since each gPA requires access to host page table) and each of the 4 levels of the guest page table (guest page table holds gPA) plus 4 references for the guest page table itself to obtain the final hPA: $4 \times 5 + 4$ references. Figure 2.9 describes the hardware page walk for nested paging.

Note that various caching techniques in today's commodity processors, like caching of page table entries in data caches [65], MMU caches [24, 30] and caching intermediate translations [28, 29] can remove some of the memory references for a TLB miss (see Sec-

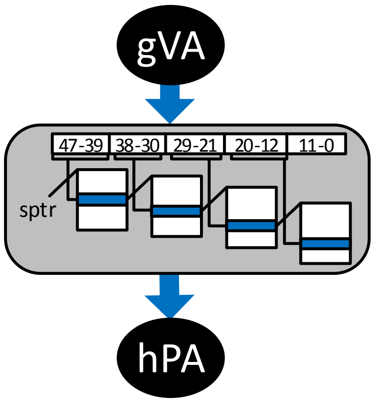

Figure 2.10: Shadow page walk using the same hardware was the native page walk.

```
shadow_walk(gVA, sptr)
return host_walk(gVA, sptr);
```

Figure 2.11: Pseudocode for shadow page walk uses host_walk as defined in Figure 2.4

tion 2.2.3). Even though the virtualized TLB miss is longer than native TLB miss, nested paging allows fast direct updates to both of the page tables without any VMM intervention.

2.2.2 Hypervisor-enabled Shadow Paging

Shadow paging is a less used software technique to virtualize memory. With shadow paging, the VMM creates a shadow page table (on demand) by merging the guest and host tables, then holds a complete translation from gVA⇒hPA. This was the only technique used before hardware support for virtualization was invented [17, 28, 78].

In the best case, the virtualized address translation hits in the TLB to directly translate from gVA to hPA with no overheads. On a TLB miss, the hardware performs a 1D page walk on the shadow page table. The native page table pointer points to the shadow page table. Thus, the memory references required for a shadow page table walk are the same as a base native walk. For example, x86-64 requires up to 4 memory references on a TLB miss for shadow paging as well as base native address translation (shown in Figure 2.10). In addition, as a software technique, there is no need for any extra hardware support for page walks. The pseudocode hardware page walk is the same as the native page walk (see Figure 2.11).

Even though the TLB misses cost the same as native execution, this technique does not allow direct updates to the page tables since the shadow page table must be consistent [17]. Every page table update requires a costly VMM intervention (VMtraps) to fix the shadow page table by invalidating or updating its entries. These VMM interventions cause significant overheads in many applications. For example, to mark a page copy-on-write by a guest OS, shadow paging requires at least two VMtraps costing 1000s of cycles: one to write to the guest page table to mark the page read-only and one to force a TLB flush. Similarly, context switches require a VMtrap for the VMM to determine the shadow page table for the incoming process. These costly VMtraps are not required for nested paging or native paging. Note that we define VMtrap latency as the cycles required for a VMexit trap and its return plus the work done by the VMM in response to the VMexit.

2.2.3 Page Walk Caches and Nested TLBs

As we saw in Section 2.1.4, page walk caches (PWCs) help reduce page walk latency by skipping some levels of page table walk in a 1D native page walk. PWCs also help skip levels of page walk in nested and shadow paging. With shadow paging as with native page walk, PWCs store the hPA as a pointer to the next level of the shadow page table and thus skip accessing a few levels of the the shadow page table walk. With nested paging, PWCs store the hPA as a pointer to the next level of the guest page table, and skip accessing some of the levels of guest page table as well their corresponding host page table accesses. The locality in PWCs help reduce a large fraction of page walk latency with nested paging since it reduces a larger fraction of memory accesses.

Apart for PWCs, nested TLBs in Intel and AMD processors store intermediate gPA⇒hPA translations which help reduce most of the host page table accesses. Usually a separate structure stores the nested TLB entries [28], but has been implemented as a physically shared TLB as well [53].

2.2.4 Dimensionality of Page Walks

Native page walk is usually referred to as a 1D page walk since there is only one dimension of the hierarchical page table depth. Whereas with a nested page walk, there are two-dimensions of the hierarchical page table depths with two page tables. Thus, it is referred to as a 2D page walk. Ideally, if an application can use the physical address directly without any page tables, then the ideal case can be termed as a 0D page walk.

With each increase in the dimensionality of page walk, the overheads associated with address translation increase. This is caused by longer latency page table walk with each increase in dimensionality of page walk. This of course impacts the performance of the application running at each level of the page table. In this thesis, we are aiming to achieve a 0D page walk for any dimensionality of page walk.

2.3 Other Related Work

The previous sections discussed state-of-the-art commercially available hardware designs to support virtual memory. This section presents other related work that comprises mostly research proposals and a few older but pivotal commercial designs.

2.3.1 Virtual Memory

When applications have very large working sets, TLBs can cause performance degradation on native machines [25, 26, 28, 31, 32, 75]. Workload running in a virtualized system exacerbate this problem [17, 21, 28, 38, 101]. Two classical approaches to reducing overheads of virtual memory are reducing TLB miss latency and reducing TLB misses.

Reducing TLB miss latency: To reduce TLB miss latency, PTEs are cached in data caches [24, 28, 62] or TLB miss latency is hidden with help of shared TLBs [31, 33]. Prefetching PTEs can also improve performance [28, 66, 67, 88]. A special structure was proposed to cache the nested translations (gPA⇒hPA) to reduce the latency of a TLB miss with virtualization [28, 29]. Translation caching [24] and Large-Reach MMU caches [30] improved performance by caching any intermediate translation. Some ISAs like SPARC use software managed TLBs and use a software-defined translation buffer (TSB) to service TLB misses faster [62].

Reducing TLB misses: Large-pages improve TLB coverage, thus reducing TLB misses [49, 51, 92, 95, 96, 97]. Most processors support multiple page sizes today [66]. However, applications and OSes have been slow to support multiple page sizes [51, 96]. Hardware support for large pages is difficult to design [24, 94, 96]. Coalescing contiguous or clustered PTEs have been shown to improve effective TLB size [81, 82]. There have been some efforts to enable large pages more effectively. Papadopoulou et al. [79] proposed a prediction mechanism that allows all page sizes to share a single set-associative TLB. Du et al. [48]

proposed mechanisms to allow huge pages to be formed even in the presence of retired physical pages.

Various software techniques help manage TLBs and reduce TLB misses. Intel Itanium had a software managed section in the TLB to pin critical address translations [45]. A recent proposal supports virtualization with software-managed TLBs [40].

The other common approach for reducing TLB misses is virtual caching [26, 66, 89, 90, 105, 106]. However, big-memory workloads often have poor cache performance, so virtual caching merely moves, but does not solve the problem. Other mechanisms also include fine-grained memory protection [57, 99, 103], but they do not reduce address translation costs. We will next discuss two approaches to reduce overheads of virtual memory in particular which we use and compare against in this thesis.

2.3.2 Multipage Mappings

Multipage Mapping approaches, such as sub-blocked TLBs [96], CoLT [82] and Clustered TLBs [81], pack multiple PTEs into a single TLB entry. Figure 2.6(b) in Section 2.1.5 compares this technique with hierarchical TLBs and huge pages discussed before. 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 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.

2.3.3 Direct Segments

Direct segments use a form of segmentation along with paging to largely eliminate virtual memory overhead for big-memory workloads on unvirtualized (native) hardware [25]. A direct segment maps one portion of a process's linear address space with a segment rather than paging. Thus, a large chunk of a contiguous virtual address space can be mapped to contiguous physical addresses with only three registers per hardware context: BASE, LIMIT and OFFSET where BASE and LIMIT are the start and end of contiguous virtual address space and OFFSET is the difference between the virtual addresses and physical addresses of the direct segment. For compatibility, the rest of the linear address space is mapped using conventional paging. On a memory reference, the processor consults the segment registers and L1 TLB in parallel, with at most one match. A virtual address V within a direct segment (BASE $\leqslant$ V $<$ LIMIT) gets translated to physical address V+OFFSET via simple addition avoiding the possibility of a TLB miss. Figure 2.12 illustrates an example address space mapping and Figure 2.6(d) in Section 2.1.5 compares direct segments with other techniques that increase TLB reach. Basu et al. show that direct segments can eliminate 99% of address translation overhead for native big-memory applications.

To expose this hardware to programs, Basu et al. propose the *primary region* abstraction, which is a contiguous chunk of virtual address space that is mapped with the same access permissions. A primary region can be mapped fully or partially by a direct segment (shown in Figure 2.12). However, direct segments are limited because they require programmer intervention and only one segment is active at once.

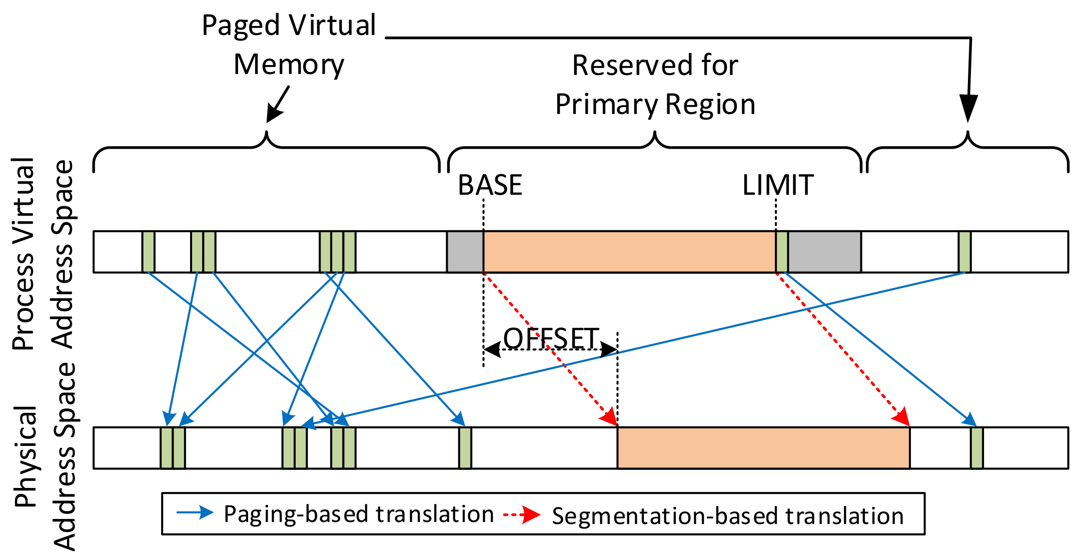
Paged Virtual Memory
Reserved for Primary Region
BASE
LIMIT
OFFSET
Process Virtual Address Space
Physical Address Space
Paging-based translation
Segmentation-based translation

Figure 2.12: Address space layout using direct segments.

2.3.4 Virtualization

Virtualization has been used since the 1970s [56] to run multiple OSes on a single machine by introducing a layer of indirection between hardware and operating systems called a hypervisor or VMM. Virtualization's renaissance began with Disco in 1996 [39] and progressed without hardware support [17, 23] (e.g., dynamic binary translation [17, 20]) and, after gaining popularity, with hardware support [28, 29].

Virtualization of memory management units followed the same software-to-hardware progression. The key software-only technique initially was shadow paging [100]. All x86-64 and ARM64 processors now support the 2D page walk in hardware [28, 29].

In addition, hardware support for virtualization has greatly reduced the number and the latency of VMM interventions [12, 28, 29]. Binary translation can reduce these interventions further [20]. But a 2013 study from VMware shows that big-memory workloads have high virtualization overheads [38].

2.3.5 Flat Page Tables for Virtual Machines

Ahn et al. [21] proposed replacing x86-64's four-level nested page table (gPA⇒hPA) with a flat one-level nested page table. This closely related work reduces the number of nested page table accesses from 4 to 1 and a 2D page walk (gVA⇒gPA & gPA⇒hPA) from 24 to 8 accesses. While this is promising for the small virtual machines they studied, it may be less suited for big-memory workloads. Our proposals goes beyond and reduce overheads higher than this work.

2.3.6 Selective Hardware Software Paging (SHSP)

Past work—selective hardware software paging (SHSP)—showed that a VMM could dynamically switch an entire guest process between nested and shadow paging to achieve the best of either technique [101]. It monitored TLB misses and guest page faults to periodically consider switching to the best mode. However, switching to shadow mode requires (re)building the entire shadow page table, which is expensive for multi-GB to TB workloads.

2.3.7 Segmentation

Segmentation has been used in various processors. The Burroughs B5000 [73] was an early user of pure segments. In general, segmentation is used without any paging other systems like the early 8086 [2], or on top of paging later, as in IA-32 [66] and MULTICS [46]. Segmentation on top of paging in x86-64 has been used with virtual machines [19]. A flat memory was also used in Blue Gene Linux for HPC workloads [107]. We use segmentation along with paging, but never for the same address. Segmentation in virtualized systems has been mentioned, but without any hardware design or evaluation [25, 60].

We covered a plethora of related work in this chapter. We will now use these as background to build our proposals in this thesis.

Chapter 3

BADGERTRAP: A TOOL TO INSTRUMENT X86-64 TLB MISSES

3.1 Introduction

This thesis deals with MMU (Memory Management Unit) research, which has been often performed with cycle-level simulators like gem5 [35]. While these simulators provide the flexibility of modeling any architectural innovation, they suffer from three drawbacks. First, TLB misses are rare events (a few per thousand instructions), which require long simulations to accurately measure their performance. Especially for big-memory workloads, a full-system simulation would take weeks, if not months, of simulation time to provide any insightful information. Second, memory requirements for running big-memory applications in a cycle-level simulator are much higher, requiring long startup times to initialize memory. Also, a single simulation point in gem5 takes at least twice as much physical memory as the workload [25]. Third, OS-level issues like timer-interrupts are usually approximately modeled, which makes simulators not good for studying systems-level issues like TLB misses. Therefore, it is challenging to use such detailed simulators to gain insight into micro-architectural techniques for improving MMU efficiency.

We address this issue with a new tool called *BadgerTrap*, which uses online instrumentation of TLB misses. It performs a higher-level analysis of a large number of TLB misses to get a better understanding of a proposed architecture. Specifically, BadgerTrap analyzes hardware or software functions that affect x86-64 TLB misses, such as new page table layouts or address translation mechanisms. These are especially important to study in big-memory workloads, where TLB misses are more frequent [25].

BadgerTrap intercepts each hardware-assisted page walk on an x86-64 TLB miss and converts it into a page fault handled specially inside the kernel with a new software-assisted TLB miss handler. We extend the handler for online analysis while running applications. BadgerTrap slows down workloads by about 2x to 40x based on their rate of TLB misses. However, BadgerTrap runs orders of magnitude faster than binary instrumentation tools like Pin [74]. The main contributions of the tool are:

(i) a novel tool that intercepts both data and instruction TLB misses, converting hardware-assisted page walks to software-assisted page walks,

(ii) extensions of software-assisted page walks to instrument and analyze x86-64 TLB misses on real hardware, which is orders of magnitude faster than full-system cycle-level simulators.

This thesis evaluates all its proposals with BadgerTrap. We describe how to extend the BadgerTrap tool to evaluate our proposals in their respective chapters. This tool was originally published in Computer Architecture News (CAN) in its September 2014 edition [52] and was developed in collaboration with Arkaprava Basu.

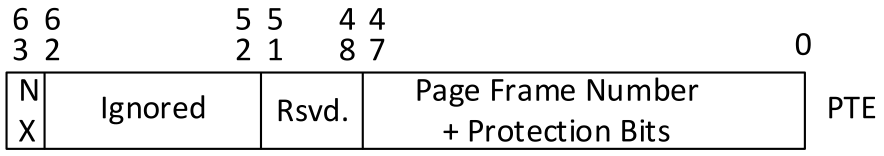

Figure 3.1: Format of a 64-bit Page Table Entry [44].

3.2 Design

This section describes the mechanism to convert a hardware-assisted page walk on an x86-64 TLB miss into a software-assisted page walk using an instrumented Linux kernel. The design has four main components:

(i) **Intercepting TLB misses** by marking PTEs as invalid/poisoned

(ii) **Software-assisted TLB miss handler** for the TLB misses being intercepted

(iii) **Attaching BadgerTrap** to processes

(iv) **Instrumenting TLB misses** to perform interesting studies on a program.

3.2.1 Intercepting TLB misses

To intercept the hardware page walker, we *poison* the PTEs at the leaves of the page table to force the system to trap rather than load a PTE. In x86-64 systems, on a TLB miss, a hardware page-table walker walks the four-level page table to load a new TLB entry for virtual address that needs translation.

BadgerTrap poisons a PTE by setting a reserved bit (one of bits 48-51) in a PTE (see Figure 3.1). The poisoned PTEs can be at L4, L3 and L2 levels of the page table to support all page sizes (4KB, 2MB and 1GB, respectively). This causes the hardware page walker

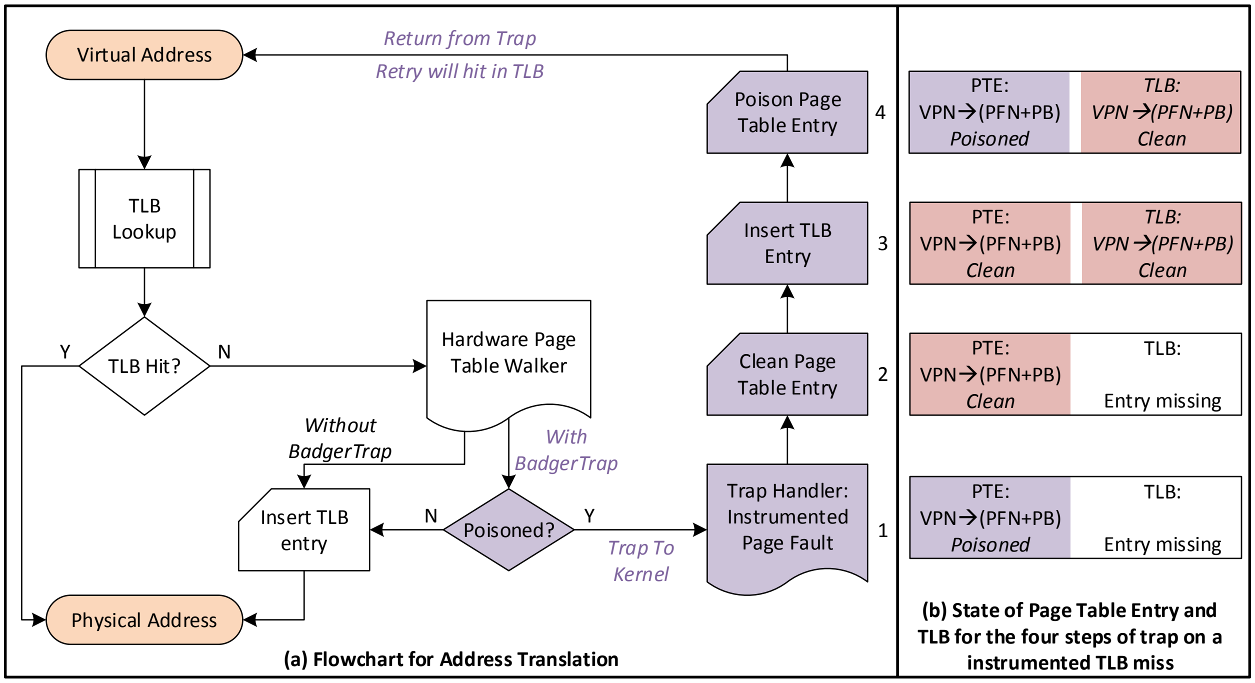

Figure 3.2: Flowchart for each translation with and without BadgerTrap along with state of PTE and TLB during an instrumented TLB miss.

to raise a page fault exception with RSVD bit set in the page fault exception flags [44]. Using a reserved bit rather than the valid bit allows the page-fault handler to quickly determine from the RSVD flag whether the fault is real or caused by instrumentation, without accessing memory. We instrument the Linux kernel to handle this exceptional page fault with a special TLB miss handler. Note that these faults occur whether the page is referenced from user mode or kernel mode, allowing BadgerTrap to track kernel-induced TLB misses on user memory.

3.2.2 Software-assisted TLB miss handler

To make forward progress, BadgerTrap handles these exceptional TLB misses by inserting a translation into the TLB. Figure 3.2 (a) shows the steps (marked 1-4) involved in handling this exceptional TLB miss. While handling the TLB miss in kernel mode, we un-poison the PTE for which the exception was raised (clear the reserved bit). We introduce the correct PTE into the TLB by referencing the page, and then poison the PTE again. This approach works since the x86-64 architecture allows the page tables and TLBs to be incoherent. Thus, the TLB can cache translations until the OS explicitly invalidates the translation. Figure 3.2 (b) shows how incoherence can exist between PTE and TLB entry with each step in the TLB miss handler.

The main trick involved in the TLB miss handler is explicitly loading both user data and instruction TLB entries for a process while in kernel-mode. To load a DTLB entry from kernel mode, BadgerTrap reads from the virtual address causing the fault. This causes the page table walker to load the (now valid) PTE into the DTLB.

Loading an ITLB entry is more involved. To introduce an ITLB entry while in kernel mode, an instruction needs to be executed on the page containing the faulting address. The kernel cannot start executing user code, because it would not regain control. Instead, we use a technique similar to Rosenblum's *context sensitive mappings* [86]. We *dynamically overwrite* the first 13 bytes of the faulting user code page by saving the original instructions and adding a jump instruction to regain control. After executing this code, we replace the original instructions in the user code page before restarting execution in user-mode.

Every core has its own MMU unit to support multithreaded and multiprogrammed workloads. Our tool works naturally with multiprogrammed workloads, but it only works with multithreaded programs for DTLB misses. This limitation on ITLB misses comes from the usage of the above technique which exposes our code patch to other threads in user-mode. We are not aware of an effective mechanism for each core to have exclusive access to a code page while it dynamically writes to it.

3.2.3 Enabling BadgerTrap

BadgerTrap can be attached to a running process by providing its process ID to the newly created system call. It can also be attached to a new process at startup by passing the program's filename to the tool. Every time a binary with that name is launched, BadgerTrap automatically attaches to it. We provide a user-mode utility that works as a wrapper and an easy to use interface.

At process startup or when attached dynamically, BadgerTrap walks the page tables and marks each leaf PTE as poisoned. In addition, as a program makes progress, whenever a physical page is allocated to the process by having a page fault, we intercept these page faults and poison the newly created PTEs. This keeps all leaf PTEs in the dynamically changing page table poisoned while the process is running.

3.2.4 Instrumenting TLB misses

From the function in which we handle these exceptional TLB misses, we can instrument the misses to perform various studies. We have access to various process-level structures and registers like the task structure, program counter, faulting virtual address and the physical page address, when we are in the software-assisted page fault handler. We discuss examples of different studies using this mechanism in the next section.

3.3 Using BadgerTrap

In this section, we discuss a few ways to use the tool to perform interesting architectural studies. We will cover two published studies that uses the same mechanism.

3.3.1 Study 1: Direct Segment

Direct segments use a form of segmentation along with paging to largely eliminate virtual memory overhead for big-memory workloads on native hardware [25]. A direct segment maps a portion of a process's linear address space with a segment rather than paging. Thus, a large chunk of a contiguous virtual address space can be mapped to contiguous physical addresses with only three registers per hardware context: BASE, LIMIT and OFFSET. For compatibility, the rest of the linear address space is mapped using conventional paging. On a memory reference, the processor consults the segment registers and L1 TLB in parallel, with at most one match (See Section 2.3.3 for details).

We used an earlier version of the tool that later evolved into BadgerTrap. The TLB misses were instrumented and split into two categories: the TLB misses that would be eliminated using the new hardware and the TLB misses that would be serviced by conventional paging. Each TLB miss was put in the respective bin based on the virtual address of the TLB miss. We developed a linear model to estimate the reduction in TLB miss handling cost using performance counters and the above information. We used this simple model to estimate performance improvement using such a hardware technique.

3.3.2 Study 2: Coalesced and Shared MMU

A coalesced MMU uses the spatial contiguity available in PTEs to coalesce them into a single entry, thereby increasing the reach of the MMU cache. In addition, a shared MMU which is shared between cores allows multiple cores to share MMU cache entries [30], thereby improving capacity of the MMU cache. Bhattacharjee recently proposed a hardware-software co-design with these two optimizations to reduce MMU overheads [30].

To evaluate such an MMU optimization, Bhattacharjee created real-system memory trace tool using an independently developed mechanism similar to BadgerTrap. The tool dumps system memory trace on allocation of a new DTLB entry. This trace has a list of distinct page references to the DTLB. The trace basically can be thought of as a trace generated by a one-entry DTLB having DTLB misses. To create a DTLB miss trace for a one-entry DTLB, DTLBs are flushed between every DTLB miss detected. They used such a memory trace in order to get an estimate of improvement in hit-rate of their new MMU design [30].

3.3.3 Study 3-5: This Thesis

BadgerTrap is used for the evaluation in the next three chapters. See Section 4.6 for how it is used to evaluate virtualized direct segments. See Section 5.5 for how it is used to evaluate agile paging. See Section 6.6 for how it is used to evaluate redundant memory mappings.

3.3.4 Performance

Since we are converting a TLB miss to a software-assisted TLB miss handler, BadgerTrap does slow down the application being analyzed. BadgerTrap, in general, slows down workloads by around 2x to 40x based on the rate on TLB misses. Binary instrumentation tools like Pin [74], which instrument all instructions, usually slows down applications by a magnitude higher than BadgerTrap.

3.3.5 Discussion

BadgerTrap, with its limited-support for ITLB, can be used to dump real-system memory trace similar to the tools like Pin [74]. BadgerTrap helps to induce a DTLB miss for every memory access by flushing both TLBs while servicing any TLB miss. This support improves upon the memory-system trace used by Bhattacharjee for his analysis (Section 3.3.2) by having a DTLB miss trace for a zero-entry DTLB instead of a one-entry DTLB using ITLB support in BadgerTrap.

BadgerTrap can also be applied to analyze TLB misses in a virtual machine by attaching BadgerTrap to a process running inside of a guest OS. It works with Linux running on both VMware- and KVM-based virtual machines.

In the current form as released, BadgerTrap prints only the count of DTLB misses, but the TLB miss handler can be instrumented to dump real-memory system trace or instrument to perform other interesting studies. The user of the tool will have to instrument the Linux kernel to perform such studies. But since the tool has streamlined the function for instrumentation, we expect the instrumentation step will be fairly easy to write.

For valdation of the tool, we compared the TLB misses measured by performance counters while running the application alone and then running them with BadgerTrap. The comparison showed that BadgerTrap captures all TLB misses faithfully and its within 1% of performance counter results. These variations can occur across running the same experiments again. The validation was performed for all page sizes in x86-64.

Some benefits of using BadgerTrap are:

1. The tool captures traces with real-systems effects along with physical addresses which other tools like Pin [74] do not.

2. The trace generation is faster than tools like Pin [74] since we only instrument memory references and not all instructions or TLB misses, which are orders of magnitude less frequent.

3. The tool captures more detailed information in the presence of additional levels of abstractions (e.g. virtual machines).

3.4 Limitations and Future Work for the Tool

This software has only been tested with Linux Kernel v3.12.13. This tool may need to be tweaked to port it to older or newer kernels. The steps provided above are generic but are only tested on an Ubuntu- and Fedora-based operating systems. We suggest that users gain some experience with kernel development before using this tool.

The mechanism may not work for a 32-bit system since their page-table entries do not have reserved bits. This mechanism has been tested on various Intel x86-64 processors. As per our knowledge, AMD processors may support reserved faults and thus the tool may work on AMD processors. The reserved bits in the PTEs may be different from that of Intel processors.

BadgerTrap can also be attached to multithreaded programs with the exception of ITLB (see Section 3.2). Even with multithreaded programs, the slowdown is not much larger than single-threaded programs since we use the distributed locking already inbuilt into the page table structure.

This tool currently does not support NUMA memory, Kernel Samepage Merging (KSM) [3] and kernel TLB misses in Linux and many more exotic features available in the Linux kernel. The tool can be easily extended to support these different memory management optimizations.

Chapter 4

VIRTUALIZED DIRECT SEGMENTS

4.1 Introduction

Virtualization has gained importance and has various benefits espcially in the cloud such as server consolidation, isolation, better resource management and fault isolation. Virtualization's benefits, however, come with overheads in processing, I/O, and memory. Fortunately, hardware advances in virtualization [28, 29, 63] (e.g., virtualizing I/O in hardware), have reduced these overheads substantially. However, overheads for virtualizing memory are high, especially for big-memory workloads. To quote a 2013 VMware paper:

We will show that the increase in translation lookaside buffer (TLB) miss handling costs due to the hardware-assisted memory management unit (MMU) is the largest contributor to the performance gap between native and virtual servers. —Buell et al., 2013 [38]

Our results corroborate Buell et al. and show that virtualization degrades performance in workloads that use substantial memory. Figure 4.1 provides a preview of the overheads associated with virtual memory for the native case (bar: 4K) and the virtualized case with the guest OS using 4KB pages and the VMM using 4KB, 2MB and 1GB (bars: 4K+4K, 4K+2M, and 4K+1G, respectively). We observe that overheads increase drastically with virtualization and remain high even with larger VMM pages. This overhead makes virtualization less attractive for big-memory workloads that reference vast memory with

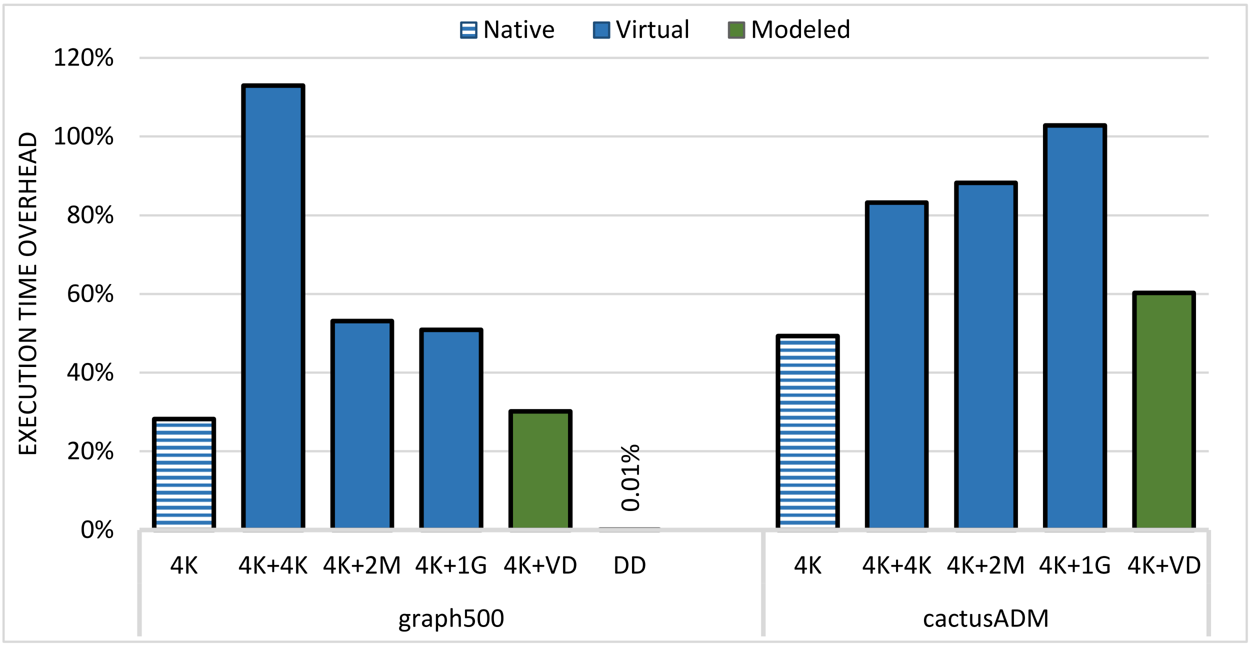

Figure 4.1: Overheads associated with virtual memory for selected workloads and configurations.

poor locality, such as key-value stores and databases. Our proposed design (bars: DD and 4K+VD) mitigates these overheads.

We propose new hardware that supports three new virtualized modes to lower the overheads of virtualized address translation. It extends direct segments [25], which were proposed to reduce TLB misses in unvirtualized (native) systems. As reviewed in Section 2.3.3, a direct segment maps most of a process's linear virtual address space with a segment, while mapping the rest with page tables.

This proposal was created in collaboration with Arkaprava Basu with me as the primary author and was origially published in the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47) 2014 [53].

Figure 4.2 depicts the two base modes (native and virtualized), one unvirtualized direct segment mode (shaded) and three new virtualized modes (shaded). Large rectangular

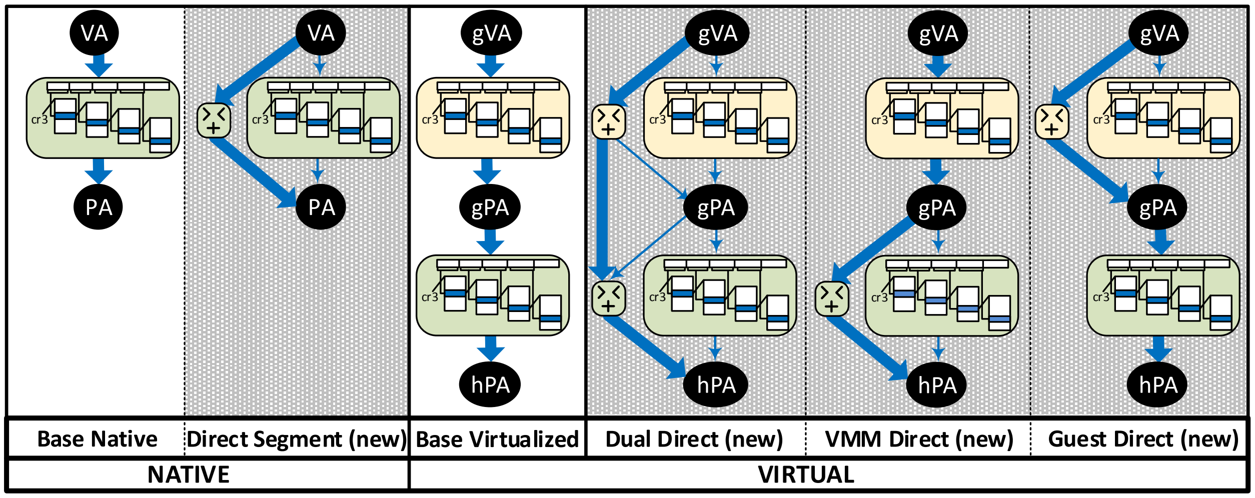

Figure 4.2: Native and virtualized address translation modes supported by proposed hardware.

boxes represent an x86-64 page table walk, small squares represent direct segment accesses (used throughout the chapter), and thick arrows depict the expected path of most translations. The left two modes show native (unvirtualized) 1D translation with and without direct segments. The right four modes are virtualized two-level translations with the existing 2D page walk (base virtualized) and our three new virtualized modes, representing different combinations of direct segments in the guest and nested address spaces. Note that Figure 4.2 shows nested translation linearly to simplify the illustration

Dual Direct mode achieves almost zero translation overheads for big-memory workloads with modest changes to the VMM, guest OS, and applications. It uses direct segments to map directly from gVA to hPA, bypassing both dimensions of virtualized address translation.

VMM Direct mode achieves near-native performance for arbitrary workloads with modifications confined to only the VMM. It uses a direct segment to map most of a guest's physical memory to host physical memory (2nd dimension: gPA⇒hPA).

Guest Direct mode provides near-native performance for big-memory workloads while using nested page tables in the VMM to facilitate services like live migration. It uses a direct segment to map most guest virtual addresses to guest physical addresses (1st dimension: gVA⇒gPA).

Unvirtualized Direct Segment mode provides identical behavior as the original direct segment proposal [25], but with less intrusive hardware.

To increase the flexibility of direct segments, we propose two novel techniques to address their limitations. First, a VM may lack contiguous physical memory needed to create a direct segment due to fragmentation. We address this with a new mechanism called self-ballooning, which uses ballooning [100] and memory hotplug [8] to create contiguous physical guest memory from fragmented guest physical memory. Ballooning removes the fragmented memory from use and hotplug adds it back as contiguous guest physical memory that can be used to create a guest direct segment. We extend self-ballooning to further increase the amount of contiguous guest physical memory by relocating memory before the x86-64 I/O gap to contiguous memory at the end. Second, we address the concern that a single faulty physical page can prevent the creation of a contiguous direct segment with an escape filter that allows holes in direct segments. The escape filter allows the OS to remap a few faulty pages within a direct segment through conventional paging. We find a 256-bit escape filter retains the performance gained by direct segments even in the presence of 16 faulty pages.

We emulate our proposed hardware and prototype our proposed software in the Linux operating system with KVM/QEMU as the VMM on x86-64. We evaluate our designs with big-memory workloads (graph500, memcached, NPB:CG), the micro-benchmark GUPS,

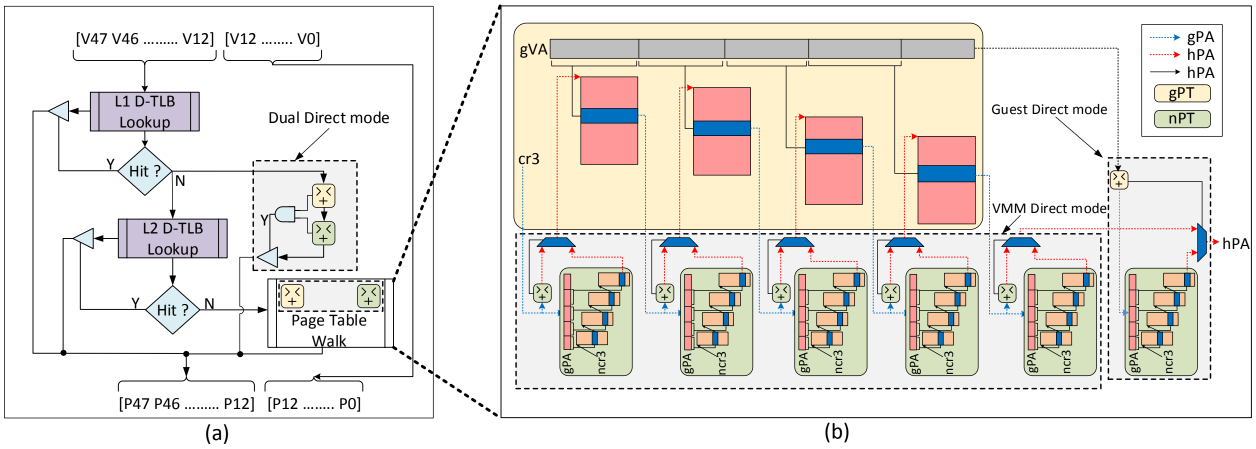

Figure 4.3: (a) Address translation flow chart (b) Steps of page walk state machine for various supported modes.

and compute workloads (SPEC 2006 and PARSEC). VMM and Guest Direct modes reduce translation overheads (page walk times) to near-native, while Dual Direct mode makes overheads negligible.

This proposal is one of the first to study the memory overhead of virtualization for virtual machines with a large amount of memory. Our contributions are:

1. a quantitative analysis of address translation overheads for large virtualized workloads,
2. a new virtualized address translation design using direct segments with three new virtualized modes for reducing overhead to make virtualization more attractive,
3. a self-ballooning technique to increase contiguity of guest physical address, and
4. an escape filter to handle physical memory faults that may exist in a direct segment.

4.2 Hardware Design and Software Support

The hardware proposed supports two levels of segment registers that can be controlled independently by the VMM and guest OS and used alone, together, or not at all. The hardware design we propose is shown in Figure 4.3. This hardware design supports three new virtualized modes, as was specified in Section 4.1. At any point in time, each guest process (address space) uses one mode. We next explain the workings of each mode using the proposed hardware, along with the software support required.

4.2.1 Dual Direct mode

Dual Direct mode seeks a zero-D (0D) page walk. It uses two layers of direct segments: one, called the guest segment, for most of the first level of address translation (gVA$\Rightarrow$gPA) and the other, called the VMM segment, for (most of) the second-level of translation (gPA$\Rightarrow$hPA). Most L1 TLB misses are translated by segment registers and hence do not need to perform a page walk (a zero-D walk), which achieves better-than-base-native execution. However, this mode suits big-memory applications with moderate guest OS and VMM changes.

Figure 4.4 shows the layout for gVA, gPA, and hPA address spaces with a guest segment in yellow and a VMM segment in green. The direct segments can be of different sizes, but here the guest segment is shown as a subset of the VMM segment.

Hardware Operation: Dual Direct performs the first level of address translation (gVA$\Rightarrow$gPA) with guest segment registers BASE_G, LIMIT_G and OFFSET_G. Next, Dual Direct performs a second level of address translation (gPA$\Rightarrow$hPA) with segment registers BASE_V, LIMIT_V, and OFFSET_V. These registers are like those of unvirtualized direct segments (Section 2.3.3),

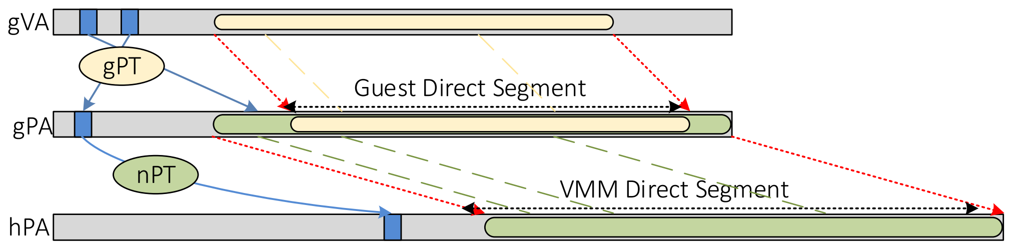

Figure 4.4: Memory layout for Dual Direct mode.

but for both levels of translation (see Figure 4.3 (a)).

A guest address can be in one of four categories in Dual Direct mode based on in which segment(s) it lies in. This is decided based on segment checks in hardware, $\text{BASE}_\text{G} \leqslant \text{gVA} < \text{LIMIT}_\text{G}$ and $\text{BASE}_\text{V} \leqslant \text{gPA} < \text{LIMIT}_\text{V}$. Table 4.1 shows the translation steps in each of the categories. In the best case, the guest address lies in both segments (Case: "Both") for which a L1 TLB miss will have a 0D page walk.

On VM-exit/entry, hardware must save/restore registers BASE_V, LIMIT_V and OFFSET_V along with other VM state.

Software Support: For Dual Direct mode, both the VMM and guest segments need modest software support. The VMM requires two key changes: (a) it must request contiguous

Steps of	Guest Virtual Address in Guest Segment or VMM Segment?			
Translation	**Both**	**VMM Segment only**	**Guest Segment only**	**Neither**
L1 TLB Hit	Translation complete	Translation complete	Translation complete	Translation complete
L1 TLB miss	$\text{hPA}=\text{gVA}+\text{OFFSET}_\text{G}+\text{OFFSET}_\text{V}$ Insert L1 TLB entry	L2 TLB lookup	L2 TLB lookup	L2 TLB lookup
L2 TLB hit	—	Insert L1 TLB entry	Insert L1 TLB entry	Insert L1 TLB entry
L2 TLB Miss	—	Invoke PTW	Invoke PTW	Invoke PTW
PTW: gVA⇒gPA	—	Walk guest page table	$\text{gPA}=\text{gVA}+\text{OFFSET}_\text{V}$	Walk guest page table
PTW: gPA⇒hPA	—	For each gPA, $\text{hPA}=\text{gPA}+\text{OFFSET}_\text{V}$ or walk nested page table	Walk nested page table	For each gPA, walk nested page table
PTW: End	—	Insert L1 TLB entry	Insert L1 TLB entry	Insert L1 TLB entry

Table 4.1: Steps in address translation of a guest virtual address in Dual Direct mode.

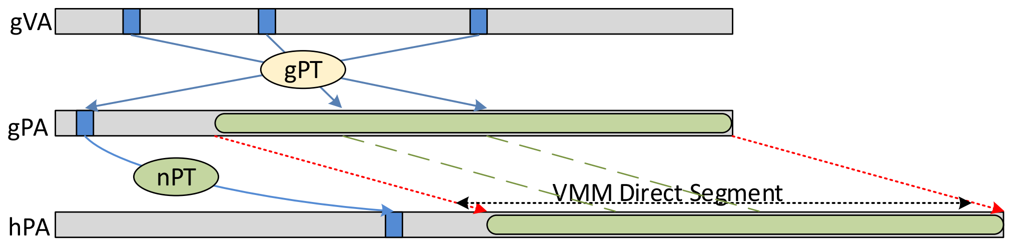

Figure 4.5: Memory layout for VMM Direct mode.

host physical memory, and (b) it must set/clear segment registers when switching between guest VMs. The value of the BASE$_V$, LIMIT$_V$, and OFFSET$_V$, registers are the start and end of the guest physical contiguous memory region, and the difference between the guest and host physical addresses of the region.

Support for guest segments for the first level of translation (gVA⇒gPA) follows unvirtualized direct segments (Section 2.3.3). They are suitable for big-memory applications with a primary region and OS support for allocating contiguous physical addresses. The guest segment register values are set per guest process and must be set during guest OS context switches. With Dual Direct, however, the OS becomes the guest OS and the physical address range is a gPA range. Like direct segments, restrictions imposed on primary regions make this less useful for compute workloads.

4.2.2 VMM Direct mode

VMM Direct mode seeks a 1D (faster than 2D) page walk with no application or guest OS changes. It uses paging for the first level of address translation (gVA⇒gPA) and a direct segment for most of the second (gPA⇒hPA). This allows applications to use a dense or sparse gVA space, and leaves the guest OS unchanged with standard paging. Figure 4.5 shows the memory layout for gVA, gPA, and hPA address spaces with a VMM segment in green.

Hardware Operation: For VMM Direct mode, BASE_G is set equal to LIMIT_G to nullify the effect of the dashed boxes labeled Dual Direct and Guest Direct mode in Figure 4.3. A guest address can be in one of two categories based on whether the guest address is in the VMM segment or not. The translation in each case is covered by the "VMM segment only" and "Neither" respectively from Table 4.1. The key advantage of VMM Direct mode is that TLB misses take only up to 4 memory accesses and 5 base and bound calculations, because guest page-table accesses (gPA) are translated with the direct segments rather than the nPT.

On VM-exit/entry, hardware must save/restore registers BASE_V, LIMIT_V and OFFSET_V along with other VM state.

Software Support: VMM Direct mode requires no application or guest OS changes, but only VMM support. VMM Direct requires only the two key VMM changes that were described for Dual Direct mode.

However, with small guest OS changes, better performance can be achieved. The x86-64 architecture has a gap in the physical address space between 3-4GB for memory-mapped I/O [2]. In Section 4.3, we describe how a simple guest OS extension can relocate memory before the gap to create a larger contiguous region above the gap. Moreover, the guest OS must allocate page tables within the VMM direct segment, which can be achieved with a guest kernel module like VMware Tools, thus improving performance.

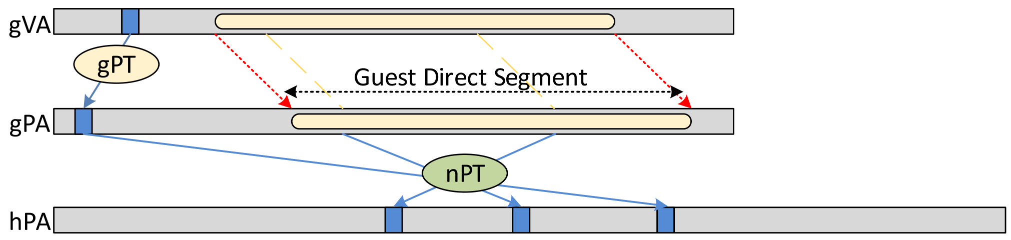

Figure 4.6: Memory layout for Guest Direct mode.

4.2.3 Guest Direct mode

Guest Direct mode seeks a 1D page walk while supporting features like page sharing and live migration that depend on 4KB nested pages. It uses a guest segment for (most of) the first level of translation (gVA⇒gPA) and nested paging for the second (gPA⇒hPA). This mode retains nested page table in the VMM while providing near-base-native speeds. It suits big-memory workloads along with small guest-OS changes. Figure 4.6 shows the memory layout for gVA, gPA, and hPA address spaces with a guest segment in yellow.

Hardware Operation: For Guest Direct mode, BASE_V is set equal to LIMIT_V thus nullifying the effect of dashed box labeled Dual Direct mode and VMM Direct mode in Figure 4.3. A guest address can be in one of two categories based on whether the guest address is in a Guest segment or not. These are covered by the cases "Guest segment only" and "Neither" respectively from Table 4.1. The key advantage of Guest Direct mode is that TLB misses make 4 memory accesses and 1 base and bound calculation, which costs closer to a native 1D page walk (4 accesses and 0 base and bound calculations) and much less than the 24-access 2D page walk.

On every guest OS context switch, hardware must save and restore BASE_G, LIMIT_G, and OFFSET_G, along with other guest process state.

Software Support: Guest Direct mode does not require any changes to the VMM, but requires the guest OS support proposed for Dual Direct mode.

4.2.4 Unvirtualized Direct Segments

Direct segment mode operates just like the original direct segments [25] but with less intrusive hardware. The key advantage of Direct Segment mode is that TLB misses make only 1 calculation, as compared to a native 1D page walk (up to 4 accesses).

This mode is supported by using only guest segment registers to translate from VA⇒PA in parallel with the L2 TLB and introduces a L1 TLB entry. This hardware support is less intrusive than the originally proposed design [25], as it performs the segment calculation in parallel with the L2 TLB lookup instead of L1 TLB lookup, where it could affect pipeline timing. The software support remains the same.

Summary: The new hardware for virtualized direct segments logically resides in two places. On the L1 TLB miss path, Dual Direct-mode hardware consults segment registers to bypass the page walk entirely. We also modify the page-walk hardware to flatten one or two dimensions of the walk based on the mode. The hardware along with software support allows switching between modes dynamically during execution.

Table 4.2 summarizes the tradeoffs among the modes. The modes can achieve faster page walks at the cost of additional hardware and software changes or restrictions. Dual Direct provides only limited support for memory overcommit, as it uses direct segments at both address translation levels. VMM Direct supports guest swapping whereas Guest Direct also supports page sharing, ballooning, and VMM swapping. All techniques can always be used for memory outside direct segments with all three modes since it uses

Properties	Base Virtualized	Dual Direct	VMM Direct	Guest Direct
Page Walk Dimensions	2D	0D	1D	1D
# of memory accesses for most page walks	24	0	4	4
# of base-bound checks for most page walks	0	1	5	1
Guest OS modifications	none	required	none	required
VMM modifications	none	required	required	none
Application category	any	big-memory	any	big-memory
Page sharing	unrestricted	limited	limited	unrestricted
Ballooning	unrestricted	limited	limited	unrestricted
Guest swapping	unrestricted	limited	unrestricted	limited
VMM swapping	unrestricted	limited	limited	unrestricted

Table 4.2: Trade-offs in various virtualized designs.

paging.

4.3 Reducing Memory Fragmentation

Guest and host physical memory fragmentation can prevent creation of direct segments at one or both levels. Moreover, the x86-64 architecture fragments the physical memory for memory-mapped I/O. We discuss few ways to reduce memory fragmentation to enable creation of direct segments.

4.3.1 Self-ballooning

To handle guest physical memory fragmentation and facilitate quick creation of guest segments, we propose a novel software optimization: self-ballooning. The goal of this technique is to provide contiguous guest physical memory quickly from fragmented free

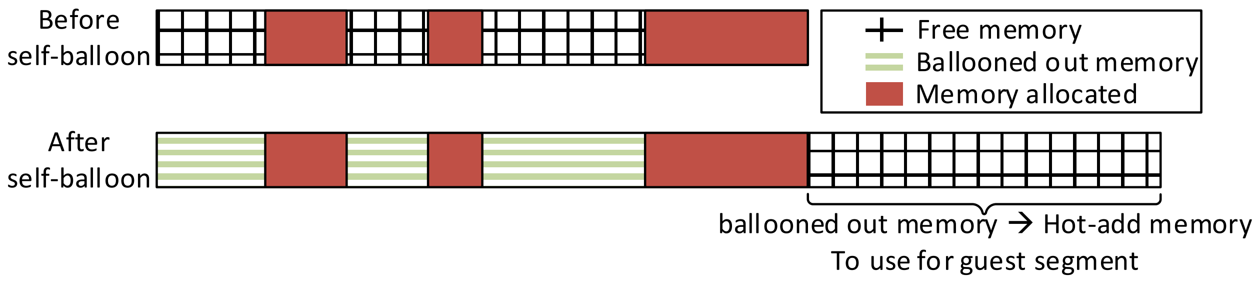

Figure 4.7: Illustration of guest physical memory with self-ballooning.

guest physical memory without the cost of memory compaction. Self-ballooning applies to Guest Direct mode and creates contiguous memory in two steps:

First, our balloon driver runs in the guest OS, and like a standard balloon driver [100], asks the guest OS for a set of pages that can be reclaimed by the VMM. The balloon driver pins and reserves this memory so it cannot be used by guest applications nor swapped out.

Second, the balloon driver passes these pages to the VMM, which uses memory hotplug to add the same amount of memory back to the VM. Memory hotplug [8] is designed for hot swapping or powering off memory chips. The newly added contiguous guest physical memory can now serve as the guest segment in the VM. Thus, the VMM can create contiguous guest physical memory from an assortment of pages. Figure 4.7 illustrates self-ballooning. While designed for Guest Direct segments, self-ballooning can also work with standard nested page tables to create more large pages in a guest OS.

4.3.2 Reclaiming I/O gap memory

A second source of fragmentation is architectural. The x86-64 architecture has an I/O gap, which is a 1GB region at the high-end of the 32-bit (4GB) physical address space reserved for memory-mapped I/O [2]. This gap splits the addresses backed by physical memory to about 3GB before the I/O gap and the rest after, and prevents a single direct segment from mapping all guest physical memory. Normally, the chipset remaps contiguous physical memory to introduce the I/O gap.

We mitigate this effect with a variation of self-ballooning using hot-unplug instead of ballooning to remove the guest physical memory. We unplug most guest physical memory before the I/O gap and extend memory by the same amount. We use hot-unplug instead of ballooning because it supports removing specific addresses (those before the I/O gap), rather than an arbitrary set chosen by the kernel memory allocator. Moving the memory allows a single direct segment to map almost all guest physical memory. More details on implementation can be found in Section 4.5.3. This technique is effective for both VMM Direct and Dual Direct.

4.3.3 Memory compaction

To address host physical memory fragmentation, we leverage the slower technique of memory compaction which slowly relocates pages and creates a VMM segment. Compaction is supported in many OSes including Linux [7]. Dual Direct and VMM Direct modes can start without a VMM segment in Guest Direct mode and Base Virtualized mode respectively. Once the compaction daemon provides enough contiguous physical memory to create a VMM segment, the VMM can create a VMM segment achieving higher performance through Dual Direct.

Summary: Table 4.3 summarizes the modes used in fragmented systems.

Applications	VM State	Modes Utilized
Big-Memory Workloads	Host physical memory fragmented	Guest Direct mode slowly converted to Dual Direct mode with host memory compaction
	Guest physical memory fragmented	Dual Direct mode with self-balloon support
	Host+Guest physical memory fragmented	Guest Direct mode with self-balloon support slowly converted to Dual Direct mode with host memory compaction
Compute Workloads	Host physical memory fragmented	Base Virtualized mode slowly converted to VMM Direct mode with host memory compaction
	Guest physical memory fragmented	Dual Direct mode
	Host+Guest physical memory fragmented	Base Virtualized mode slowly converted to VMM Direct mode with host memory compaction

Table 4.3: Various modes utilized in fragmented systems.

4.4 Escape Filter

Commodity OSes commonly put faulty pages on a badpage list to prevent their use [11, 61, 98]. However, with direct segments, a single bad page can prevent creation of a large direct segment. We introduce a novel hardware technique, an *escape filter* that allows holes in direct segment. If an address is in a hole, it "escapes" segment-based translation to use conventional paging. Hence, the VMM or OS can still use paging to remap escaped pages to functioning memory.

We implement the escape filter using a hardware Bloom filter that is checked in parallel with the VMM's segment registers (in Dual or VMM direct modes) and the guest segment registers (in Direct Segment mode). In Guest Direct mode, the VMM still uses nested pages tables and can remap faulty pages. The VMM (or OS for Direct Segment mode) adds escaped pages to the Bloom filter and creates PTEs to map those pages. As the Bloom filter may have false positives (non-escaped pages that are falsely considered escaped), the VMM must create mappings for these pages as well. With an escape filter, an address is translated with direct segments if it lies within the direct segment, *but not* the filter. The filter is part of the VM context state and must be saved/restored with VMM's segment registers. However, it can be small: to tolerate 16 faulty pages, we show that a 256-bit filter has almost zero overhead from false positives. The escape filter can also implement a limited number of pages with different protection, such as guard pages. For such uses, it may be useful to have escape filters at both levels of translation so the guest OS can escape pages as well.

4.5 Prototype Implementation

We design our prototype system for the Linux operating system (x86-64) with LTS kernel v3.12.13 using QEMU v1.4.1 with KVM as the VMM. Our implementation has three pieces: (1) allocating contiguous physical memory, (2) emulating segmentation behavior, and (3) prototyping self-ballooning.

4.5.1 Contiguous Allocation

To allocate contiguous physical memory, the OS reserves memory immediately after startup. We target machines running a stable set of long-lived virtual machines allowing us to use the mechanism without relying on expensive compaction. The size of the virtual machine and memory requirements of big-memory applications [25] are often known a priori through their configuration parameters and allow the application to reserve the required memory.

4.5.2 Emulating Segments

The hardware design described in Section 4.2 requires new hardware support. To evaluate this design on current hardware, we follow a previously proposed technique [25] and emulate direct-segment functionality by mapping segments with 4KB pages. We modify the Linux page fault handler both in the VMM and guest OS. These changes identify page faults to direct segments, and compute physical addresses using segment offset. These computed addresses are added to the respective page tables by the fault handler. Thus, direct segments are mapped using dynamically computed PTEs. This approach provides a functionally correct implementation of our designs on current hardware. However, it does not provide any performance improvement without new hardware. As described in Section 4.6, we count events to predict performance with new hardware.

4.5.3 Self-Ballooning Prototype

We implemented self-ballooning in KVM. Each guest VM has an associated KVM process, which runs as a userspace process on the host OS. The guest physical addresses of a VM are mapped on to the host virtual addresses of the KVM process, and the host Linux maps host virtual addresses of the KVM process to host physical addresses. Figure 4.8 shows a typical address space mapping in x86-64 using KVM. The KVM kernel component, called the *KVM module*, creates nested page tables by computing combined gPA⇒hVA⇒hPA translations. The mapping gPA⇒hVA is handled through memory slots. A memory slot is a contiguous range of guest physical addresses that are mapped to contiguous virtual memory in the KVM process. There are only two large slots in KVM: one between 0-4GB,

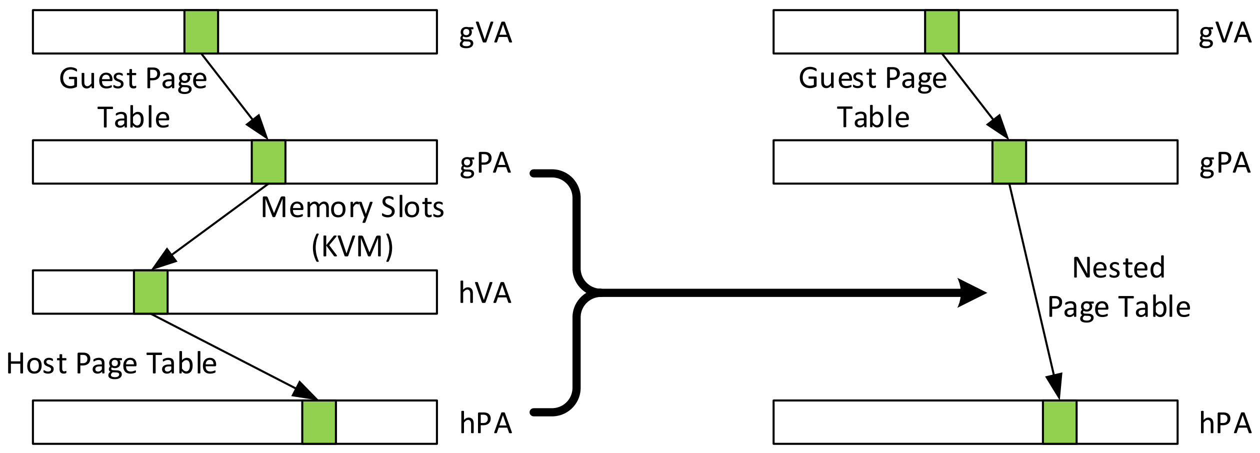

Figure 4.8: KVM memory slots to nested page tables.

and another for 4GB and beyond.

Fragmented memory: We modify QEMU-KVM [72] and the virtio balloon driver [5] to prototype self-ballooning with the KVM hypervisor. KVM currently does not support hot-adding memory to guest OSes. Instead, we extend the second KVM slot by the largest amount of memory (96GB for our machine) that can be used for self-ballooning when required. This extra guest physical memory is ballooned out during startup and cannot be used by the guest OS.

The guest OS invokes the modified balloon driver when it cannot create a guest segment due to fragmentation. The driver removes the required amount of memory and provides that to the VMM. The VMM in return informs the driver to release the memory from the reserved portion of guest physical memory. The guest OS can now create a guest segment from the newly released guest physical memory.

I/O gap: We modify the guest OS to remove as much memory as possible from the first KVM slot using hotplug [8]. Using hotplug is similar to ballooning, and causes the guest OS to ignore the removed addresses. We extend the second KVM slot by the same amount of memory. Our experiments show that 256MB is enough to boot Linux correctly and the rest (3.1 GB) can be removed from the first KVM slot. This gives us a long address range in the gPA starting at 4GB that can be mapped using a single segment to hPA, and a small 256MB range for the kernel mapped using pages.

Memory compaction: We use the memory compaction daemon present in Linux [7] to aggressively perform compaction when required to create a direct segment in the host OS.

4.6 Evaluation Methodology

We evaluate the proposed hardware using VMM and kernel modifications and hardware performance counters since workload size and duration makes full-system simulation less appropriate. We use the counters to measure the number of TLB misses in native (M_n) and virtual (M_v) environments, and the page walk cycles spent on TLB misses. This provides us with page walk cycles spent per TLB miss (C_n and C_v, respectively) for each program. We modify the guest OS kernel to capture all TLB misses using BadgerTrap (Chapter 3), a tool that instruments all DTLB misses, as these DTLB misses benefit from our proposed hardware.

We instrument the guest OS to extract gVA and gPA for a DTLB miss to determine if the address lies in a VMM or guest segment. We classify the miss depending on the mode used to calculate DTLB misses affected by a direct segment.

Native/Virtualized baseline: We run the workloads (Table 4.4) to completion on native hardware and in a virtual machine described in Table 4.5 and use Linux perf [10] to collect the performance counter data. By fixing the amount of work done by the programs, we

Workload	Description
graph500	Generation, compression and breadth-first search (BFS) of very large graphs, as often used in social networking analytics and HPC computing.
memcached	In-memory key-value cache widely used by large websites, for low-latency data retrieval.
NPB:CG	NASA's high performance parallel benchmark suite. CG workload from the suite.
GUPS	Random access benchmark defined by the High Performance Computing Challenge.
SPEC 2006	Compute single-threaded workloads: cactusADM, GemsFDTD, mcf, omnetpp (ref inputs)
PARSEC 3.0	Compute multi-threaded workloads: canneal, streamcluster (native input set)

Table 4.4: Workload discriptions.

Native System	
Processor	Dual-socket Intel Xeon E5-2430 (SandyBridge) 6 cores/socket, 2 threads/core, 2.2 GHz
Physical Memory	96 GB DDR3 1066MHz
Operating System	Fedora-20 (Linux LTS Kernel v3.12.13)
L1 Data TLB	4KB: 64 entries 4-way associative 2MB: 32 entries 4-way associative 1GB: 4-entry fully associative
L2 TLB	4KB: 512 entries 4-way associative

Virtualized System	
VMM	QEMU (with KVM) v1.4.1, 24vCPUs
Physical Memory	85GB
Operating System	Fedora-20 (Linux LTS Kernel v3.12.13)
EPT TLB/NTLB	Shares the TLB (no separate structure)

Table 4.5: Description of native and virtualized systems.

compare cycles across different configurations. For each proposed mode, we developed linear models to predict its performance (Table 4.6).

Direct Segment: To compare with native direct segments, we developed a model to determine the unvirtualized performance. We find the fraction of TLB misses (F_{DS}) that lie in the direct segment [25], which would be eliminated.

VMM Direct/Guest Direct: We determine the fraction of TLB misses that lie in the respective direct segment (F_{VD} or F_{GD}). This fraction of TLB misses would spend near-native cycles per TLB miss. We estimate the cycles per TLB miss of this fraction as (C_n+Δ) where Δ represent the overhead of performing base-bounds check in addition to cycles per TLB miss on native hardware (C_n). As an estimate, we use 1 cycle per base-bound check, thus Δ_{VD}=5 for VMM Direct and Δ_{GD}=1 for Guest Direct modes. The rest of the TLB misses would suffer C_v cycles per TLB miss due to 2D page walk.

Dual Direct: We split the TLB misses in 4 ways according to Table 4.1:

1. The fraction of misses that lie in both direct segments (F_{DD}). These page walks are eliminated by Dual Direct.

2. The fraction of misses that lie only in VMM segment and not in guest segment (F_{VD}). These misses are sped up by VMM Direct mode.

3. The fraction of misses that lie only in guest segment and not in VMM direct segment (F_{GD}). These misses are sped up by Guest Direct mode.

4. The rest of the TLB misses suffer C_v cycles per TLB miss for the 2D page walk.

The cycles per TLB miss for Dual Direct Design can be calculated by adding the above four categories. Since we cannot measure cycles spent accessing L1 and L2 TLBs, our model

Design	**Model**
Direct Segment	C_n*(1-F_{DS}) * M_n
Dual Direct	[(C_n+Δ_{VD})*F_{VD} + (C_n+Δ_{GD})*F_{GD} + C_v*(1-F_{GD}- F_{VD}-F_{DD})]*M_n
VMM Direct	[(C_n+Δ_{VD})*F_{VD} + C_v*(1-F_{VD})]*M_n
Guest Direct	[(C_n+Δ_{GD})*F_{GD} + C_v*(1-F_{GD})]*M_n

Table 4.6: Linear model for cycle spent on page walk.

does not account for improvements due to faster L2 hits when using the Dual Direct design or pollution of the L2 TLB from pages in a direct segment.

In all our experiments for this chapter, we turned all prefetchers off to reduce variations across our experiments. This methodology mostly improves overall performance of the application without much impact on the page walk latency and thus shows higher overheads of page walk in a realistic setting. However, for a more conservative estimate of overheads, this methodology was discontinued in the later chapters and all prefetchers were turned on.

4.7 Cost of Virtualization

We show that the cost of virtualization can be very high and corroborate some of the findings of Buell et al. [38]. Similar earlier studies used much smaller virtual machines running desktop applications [21, 29]. Our study quantifies how applications behave in large virtualized environments.

Our experimental setup is as follows:

- We use several multithreaded big-memory workloads, a GUPS micro-benchmark, and some SPEC2006 and PARSEC workloads (Table 4.4). The big-memory workloads were executed with 60-75GB datasets on virtual machines with 80GB of guest physical memory.

- We run the workloads on an x86-64 system (Table 4.5), both natively and in a Linux KVM virtual machine.

- We examine four native configurations. For big memory workloads, it is straightforward to have big-memory applications explicitly request 4KB, 2MB, or 1GB pages. Since SPEC and PARSEC are less suited to these changes, we use 4KB pages and enable transparent hugepages (THP) [15], which seeks to dynamically promote aligned groups of 512 4KB pages to a 2MB page.

- We examine eight virtualized configurations that vary both guest and VMM page sizes (e.g., 4K+2M means the guest uses 4KB pages and the VMM uses 2MB pages).

Figure 4.9 and Figure 4.10 present execution time overheads for address translation for base-native (hashed bars) and virtualized (solid). If an execution E runs in time T_E, we calculate its address-translation overhead as $(T_E — T_{2Mideal})/T_{2Mideal}$, where $T_{2Mideal}$ is the same benchmark's native execution time with 2MB pages minus the time the 2MB run spends in page table walks. TLB misses may be overlapped with other processor stalls, so for workloads with very high miss rates, subtracting page walk time from total execution time may be inaccurate. We try to minimize this effect by using as our base execution time, the 2MB results ($T_{2Mideal}$), but cannot remove it completely. Note that execution times include the effects of all TLBs and page walk caches. The GUPS micro-benchmark uses the scaled righthand y-axis and is shaded separately.

We make the following observations:

1. *Native address translation overheads with 4KB pages can be high and grow drastically when virtualized.* [e.g., the overheads for graph500 goes up from 28% in native to 113% under virtualization (bar 4K vs 4K+4K)]. The geometric mean increase with virtualization is 3.6x.

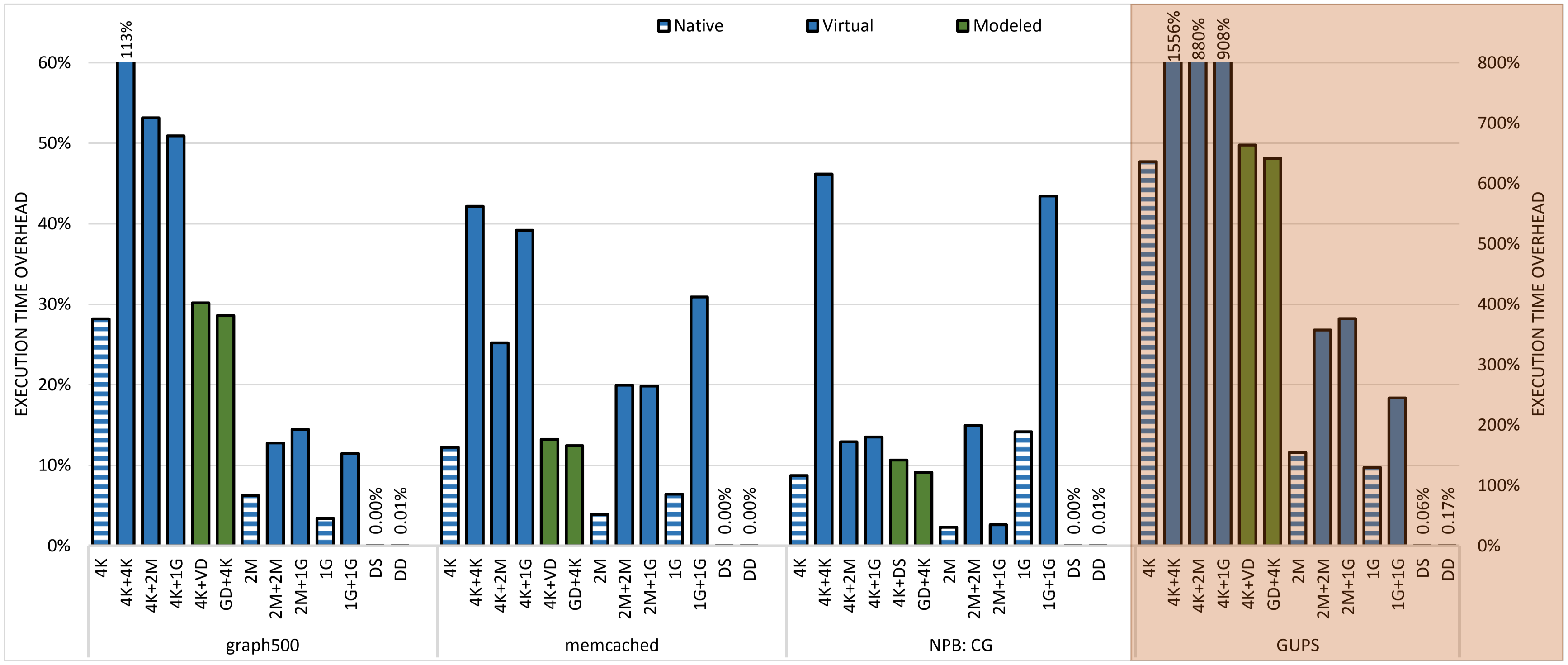

Figure 4.9: Virtual memory overhead for each configuration per big-memory workload.

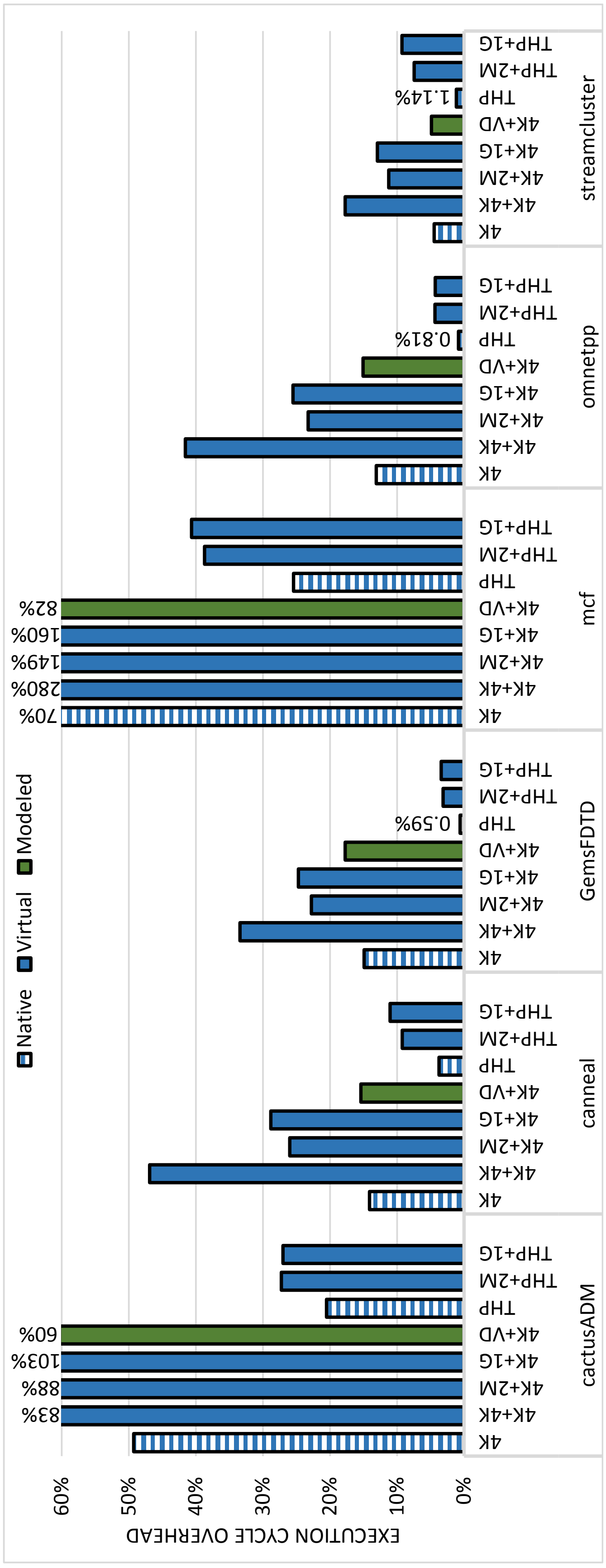

Figure 4.10: Virtual memory overhead for each configuration per compute workload.

2. *Virtualization overheads can be reduced by using 2MB pages to map gPA to hPA at the VMM. However, overheads can still be large. Even with 2MB pages at both levels of translation, overheads are substantially higher than native execution with 2MB pages* (e.g., bar 4K+2M: 53% vs bar 4K+4K: 113% and bar 2M: 6% vs. bar 2M+2M: 13% for graph500).

3. *Even with 1GB pages at guest OS and VMM, the overhead does not reduce greatly as compared to 2MB pages and are higher than native execution with 1GB pages. In addition, using 1GB pages in the VMM can also hurt performance due to limited 1GB TLB entries* (e.g., for graph500, bar 1G: 3% vs. bar 1G+1G: 11% and 13% overhead with bar 2M+2M and 14% with bar 2M+1G).

4. *Similar trends are observed in compute workloads.* CactusADM and mcf have high overheads even with transparent huge pages (THP).

These observations demonstrate that many workloads suffer substantial virtualization overheads, rendering virtualization less attractive today. Moreover, we conjecture that overheads will get considerably worse with larger datasets and virtual machines [25]. This analysis corroborates a recently published study from VMware [38].

4.8 Evaluation of New Design

In this section we discuss the various benefits of our proposed hardware with its various modes of operation.

4.8.1 Performance Analysis

Performance benefits: Figure 4.9 and Figure 4.10 depict execution time overheads for address translation for our new designs in green. Recall that our models are pessimistic due to our assumption of flat $\Delta_{VD} = 5$ cycles and $\Delta_{GD} = 1$ based on the number of base-and-bound calculations. This overhead will be reduced by techniques like translation caching [24] and shared MMU caches [30] that store intermediate translations by reducing the computations required.

From Figure 4.9 and Figure 4.10, we conclude:

1. *For our big-memory and compute workloads, VMM Direct achieves overheads close to native execution.* [e.g., graph500 suffers 30% (bar 4K+VD) overhead, close to the native overhead of 28% (bar 4K). Overall, VMM Direct is only 2% slower than native execution (geo mean)].

2. Similarly, *Guest Direct is able to achieve native performance for big-memory workloads* (bar 4K+GD).

3. *Dual Direct achieves negligible address translation overheads for big-memory workloads, similar to unvirtualized direct segments.* Dual Direct mode reduces address translation overheads to at most 0.17% (bar DD).

The performance benefits of VMM Direct are further enhanced by using 2MB (bar 2M+VD) or 1GB pages (bar 1G+VD) or THP to translate from gVA to gPA. We do not evaluate these due to lack of support for large pages in our prototype. Guest Direct can also be enhanced similarly.

Performance Breakdown: We analyzed two key factors affecting the translation overheads: 1) number of TLB misses and 2) avg. cycles spent per TLB miss.

We make two observations about execution under virtualization vs. executing natively. First, we expect the TLB misses for a given application to remain the same across execution under virtualization and native execution. However, *we found that virtualization can lead to a significant increase in the number of TLB misses.* For example, TLB misses 4K+4K increased by 1.38x for graph500, 1.62x for memcached, 1.41x for GUPS, 1.33x for canneal, and 1.29x for streamcluster. The extra misses occur because nested TLB entries (gPA⇒hPA) share the same physical structure as the normal TLB entries, reducing the effective capacity of the TLB. We confirmed the behavior with a microbenchmark. Second, *as expected for all workloads, the average cycles per TLB miss grows significantly with virtualization due to 2D page walks.* For example, it can be as high as 3.5x for NPB:CG with 4K+4K. On average, cycles-per-miss increase 2.4x, 1.5x, 1.6x for 4K+4K, 4K+2M, and 4K+1G, respectively.

With our proposed hardware, we find that VMM Direct and Guest Direct achieve cycles per TLB miss close to native execution. A TLB miss costs only 13% higher (on average) with VMM direct and 3% higher (on average) with Guest Direct compared to native 4K. In contrast, Dual Direct gets most of its benefits from reduction in L2 TLB misses (99.9% reduction in L2 TLB misses).

4.8.2 Energy Discussion

Alternate address translation modes affect system energy in two ways. Most importantly, if the mechanism reduces execution time by some percentage X, it can reduce wholesystem static energy by about X%. For example, Dual Direct reduces execution time by 11-89% compared to 4K+2M pages (Figure 4.9 and Figure 4.10) across our benchmarks and thus reduces system static energy by a similar fraction.

Second, the translation mechanism itself uses energy, (e.g., 20-38% of L1 cache dynamic energy [26] and a smaller fraction of whole system energy). Page-based address translation uses dynamic energy to: (a) access the L1 TLB, (b) on an L1 TLB miss, accesses the L2 TLB, and (c) on an L2 TLB miss, accesses the page walker and MMU cache.

Our new virtualized design: (a) leaves the L1 TLB access unchanged; (b) on an L1 TLB miss, accesses the L2 TLB as well as small virtualized direct-segment hardware; and (c) on a L2 TLB miss and on a miss in the both direct segments, accesses the modified page walker and MMU cache. We expect the reduction in term (c) dominates the small cost increase to term (b), thus potentially reducing address translation dynamic energy as compared to virtualized baseline.

The original direct segment design uses energy to: (a) access the L1 TLB as well as small direct-segment hardware, (b) on a L1 TLB miss and on a miss in the direct-segment, accesses the L2 TLB, and (c) on L2 TLB miss, accesses the page walk hardware and MMU cache. If the reduction in the term (b) dominates the small cost increase to term (a), then the original direct segment hardware further improves on address translation dynamic energy compared to the new virtualized design.

If this improvement is important, our design can be modified to perform the base-bound check for Dual Direct with its two comparators in parallel with L1 TLB lookup. We do not advocated this design, because it affects the timing critical L1 TLB hit path (a drawback of the original design).

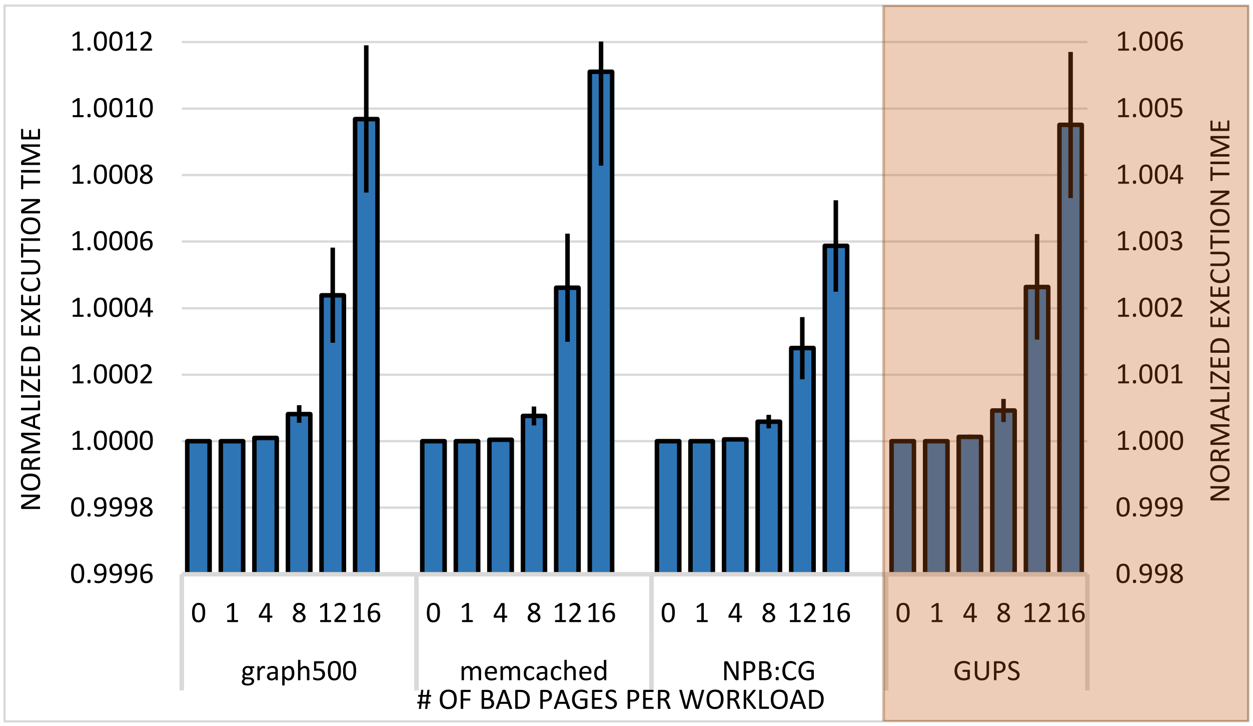

Figure 4.11: Normalized execution time for big-memory workloads in presence of bad pages.

4.8.3 Escape Filter

Our proposed escape filter enables a few pages within a direct segment to escape to page-based translation. We use this mechanism to retain most of a direct segment's performance benefits even when 1-16 pages have hard faults.

We studied the impact of using a 256-bit (32-byte) hardware parallel bloom filter with four H3 hash functions [87]. For each number of bad pages (1-16), we ran each application with 30 different random sets of bad pages. Figure 4.11 depicts execution time overhead (compared to Dual Direct mode with no bad pages), as well as 95% confidence intervals.

Dual Direct mode retains almost all its performance benefits even with some hard faults. With a pessimistic 16 faults, execution impact is less than 0.06% (except microbenchmark GUPS

0.5%). We observe similar trends with compute workloads running in VMM Direct mode.

4.8.4 Shadow Paging: An Alternative

Shadow paging offers a way to eliminate 2D page walks, but we observe that it works well for only some of our workloads, while our new design works for all. Recall that with shadow paging (Section 2.2.2), the VMM uses the guest page table (gVA⇒gPA) and nested page table (gPA⇒hPA) to build a shadow page table (gVA⇒hPA) walked by the hardware, and changes to guest or host page tables incur substantial performance overheads.

We use shadow paging by disabling extended page tables in KVM. We measure the slowdown in execution time compared to native execution for shadow paging using 4KB and 2MB pages (both guest and VMM).

We observe that our workloads fall in two categories:

1. Workloads for which shadow paging incurs high virtualization overheads: memcached (4K: 29.2%, 2M: 11.1%), GemsFDTD (4K: 12.2%, 2M: 4.9%), omnetpp (4K: 8.7%, 2M: 3.4%), and canneal (4K: 6.63%, 2M: 2.5%). The increase in execution time corresponds to extra VMexits to keep shadow page tables coherent.

2. Workloads for which shadow paging incurs relatively low overheads. For all other workloads, we observed slow-down of less than 5% for both page sizes.

Shadow paging does well due to the static nature of memory allocation in workloads from the second category. For workloads with frequent memory allocations and deallocations (first category), shadow paging provides poor performance due to frequent updates to guest page tables [101].

In contrast, VMM Direct allows page table updates to proceed without any VMM intervention. Thus, our techniques provide near-native performance for both sets of workloads. Shadow paging can be up to 29.2% slower, whereas VMM Direct is only up to 7.3% slower than the native execution. With small guest OS and application changes, Dual Direct provides much lower overhead than shadow paging.

4.8.5 Content-Based Page Sharing

Content-based page sharing saves memory for compute workloads, but we observe that it provides less benefit for our big-memory workloads. Content-based page sharing scans memory to find pages with identical contents. When such pages are found, the VMM can reclaim all but one copy and maps the others using copy-on-write [100].

We studied the impact of content-based page sharing, since VMM segments preclude page sharing. We coscheduled two smaller instances (40 GB) of KVM, each running one of our big-memory workloads (all possible pairs) to measure the potential memory saving from page sharing.

We observed that page sharing does not save more than 3% memory for our big-memory workloads since the bulk of memory is for data structures unique to the workload. These include OS code pages that can be easily shared with our modes, as they are mapped with pages. Thus, restricting page sharing may be less important when virtualizing big-memory workloads than others.

For compute workloads, earlier studies have shown page sharing to be useful when there are large numbers of VMs.

4.9 Summary

This proposal brings low-overhead virtualization to workloads with poor memory access locality. This is achieved with three new virtualized modes that improve on 2D page walks and direct segments. In addition, we propose two novel optimizations that greatly enhance the flexibility of direct segments. This design can greatly lower memory virtualization overheads for big-memory and compute workloads.

Chapter 5

AGILE PAGING

5.1 Introduction

The previous chapter on *virtualized direct segments* brings low-overhead virtualization by using direct segments to bypass paging in widely used hardware support — nested paging— but ignored a less used software technique—shadow paging. Shadow Paging provides an opportunity to reduce TLB miss latency while retaining all the benefits of paging. Nested and shadow paging provide different tradeoffs while managing two-levels of translation for robust virtualization support. Unfortunately, virtualized direct segment required substantial hardware and operating system changes which made its adoption harder. With this proposal, the goal is to provide low-overhead virtualization without with more modest hardware and VMM support as compared to virtualized direct segments.

Table 5.1 summarizes the trade-offs provided by the two techniques to virtualize memory as compared to base native. With current hardware and software, the overheads of virtualizing memory are hard to minimize because a VM exclusively uses one technique or the other. Nested and shadow paging make different tradeoffs.

First, the widely used hardware technique called nested paging [28] generates a TLB entry (gVA⇒hPA) to map guest virtual address directly to host physical address enabling fast translation (Section 2.2.1). On a TLB miss, hardware performs a long-latency 2D page walk which walks both page tables. For example, in x86-64, TLB misses require up to 24

	Base Native	Nested Paging	Shadow Paging	Agile Paging
TLB hit	fast (VA⇒PA)	fast (gVA⇒hPA)	fast (gVA⇒hPA)	fast (gVA⇒hPA)
Max. memory access on TLB miss	4	24	4	~(4—5) avg.
Page table updates	fast direct	fast direct	slow mediated by VMM	fast direct
Hardware support	1D page walk	2D+1D page walk	1D page walk	2D+1D page walk with switching

Table 5.1: Trade-off provided by both memory virtualization techniques as compared to base native. Agile paging exceeds best of both worlds.

memory references [28] as opposed to a native 1D page walk requiring up to 4 memory references. However, this technique benefits from fast direct updates to both page tables without VMM intervention.

Second, the lesser-used technique called shadow paging [100], which was used before hardware support was available, requires the VMM to build a new *shadow page table* (gVA⇒hPA) from both page tables (Section 2.2.2). It points standard paging hardware to the shadow page table (sPT), so that TLB hits perform the translation (gVA⇒hPA) and TLB misses do a fast native 1D page walk (e.g., 4 memory references in x86-64). However, page table update requires VMM to perform substantial work to keep the shadow page table consistent [17].

Past work—*selective hardware software paging (SHSP)*—showed that a VMM could dynamically switch an entire guest process between nested and shadow paging to achieve the best of either technique [101]. It monitored TLB misses and guest page faults to periodically consider switching to the best mode. However, switching to shadow mode requires (re)building the *entire* shadow page table, which is expensive for multi-GB to TB workloads.

We take inspiration from and extend this approach with *agile paging* to *exceed* the best of both techniques. Intuitively, most of the updates to a hierarchical page table occur at the lower levels or leaves of the page table. With that key intuition, agile paging starts virtualized page walk with the shadow paging for stable upper levels of page table and optionally switches within the same page walk to nested paging for lower levels of page table, which receive frequent updates. With this agile page walk, guest virtual address space to use both shadow and nested paging *at the same time* with varying degree of nesting and allows switching from one mode to the other in the middle of a page walk. Agile paging reduces the cost of a TLB miss since most of the TLB misses are handled fully or partially with shadow paging and reduces the costly VMM interventions by allowing fast direct updates to the page tables. Table 5.2 shows varying degrees of nesting and memory references for page walks in x86-64 depending on when the switch from shadow to nested paging occurs. Our evaluation in Section 6.7 shows that agile paging requires fewer than 5 memory references per TLB miss on average.

Agile paging goes beyond SHSP [101] so a process can concurrently use nested and shadow paging for different address regions (and even different levels of a single translation) at a cost of modest hardware changes. One can think of SHSP as a temporal solution since it multiplexed the two techniques in time, while agile paging is temporal and spatial, where spatial may grow in importance with increasing memory footprint.

The initial feasibility analysis of this proposal was done in collaboration with Sujith Surendran and the work was originally published with me as primary author in the 43rd ACM/IEEE Symposium on Computer Architecture (ISCA-43) 2016 [54].

Agile paging builds on existing hardware for virtualized address translation, requiring only a modest hardware change that switches between the two modes. In addition, to

Levels of Page Table	Base Native	Nested Paging	Shadow Paging	Agile Paging
PTptr: Page table pointer	0	4	0	0 or 4
L4: Page table level 4 entry	1	5	1	1 or 5
L3: Page table level 3 entry	1	5	1	1 or 5
L2: Page table level 2 entry	1	5	1	1 or 5
L1: Page table entry (PTE)	1	5	1	1 or 5
All	4	24	4	4-24

Table 5.2: Number of memory references with varying degree of nesting provided by agile paging in a four-level x86-64-style page table as compared to other techniques.

further reduce VMM interventions associated with the shadow technique within agile paging, we propose two optional hardware optimizations. Similarly, VMM support for agile paging builds upon existing support and requires modest changes.

Figure 5.1 shows address translation techniques for base native, nested paging, shadow paging, and our technique of agile paging. The numbers in the figure show the chronological order in which different levels of the page table structures are accessed on a TLB miss. The page tables are hashed in shadow paging since the hardware has no access to the guest or host page tables. Agile paging is color coded to show two of the options available from Table 5.2. The black colored path shows shadow paging in agile paging. With the blue colored escape path, agile paging switches the hardware from shadow paging to nested paging for the leaf-level of page table, requiring up to 8 memory accesses per translation. The switch from shadow to nested can be performed at any level of the page table (not shown).

We emulate our proposed hardware and prototype our proposed software in KVM on x86-64. We evaluate our design with variety of workloads and show that our technique

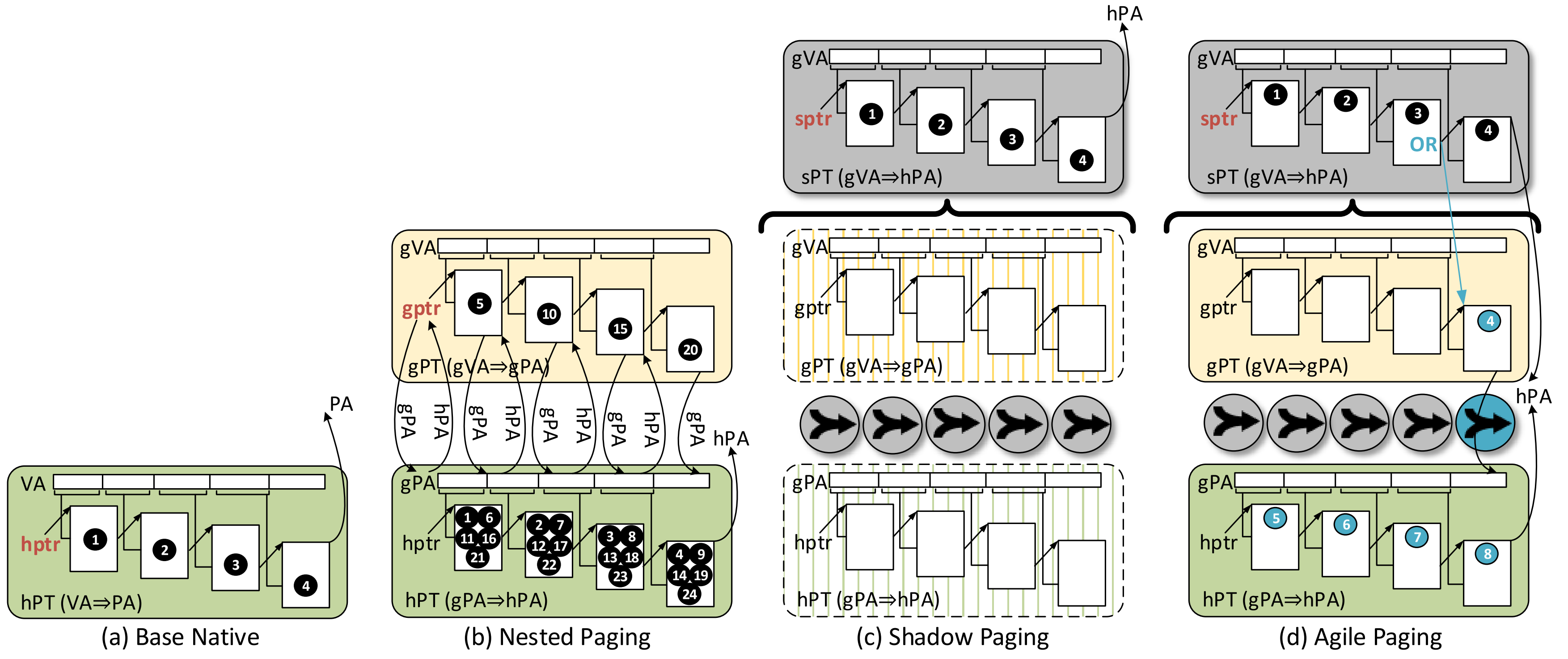

Figure 5.1: Different techniques of virtualized address translation as compared to base native. Numbers indicate the memory accesses to various page table structures on a TLB miss in chronological order. The merge arrows denotes that the two page tables are merged to create shadow page table. Colored merge arrows with agile paging denotes partial merging at that level. The starting point for translating an address is marked in bold and red.

improves performance by more than 12% compared to the best of nested and shadow paging.

In summary, the contributions of our work are:

1. We propose a mechanism *agile paging* that simultaneously combines shadow and nested paging to seek the best features of each within a single address space.
2. We propose two optional hardware optimizations to further reduce overheads of shadow paging.
3. We show that agile paging performs better than the best of shadow and nested paging.

5.2 Agile Paging Design

We propose *agile paging* as a lightweight solution to the cost of virtualized address translation. We observe that shadow paging has lower overheads than nested paging, except when guest page tables change. Our *key intuition* is that page tables are not modified uniformly: some regions of an address space see far more changes than others, and some levels of the page table, such as the leaves, are updated far more often than the upper-level nodes. For example, code regions may see little change over the life of a process, whereas regions that memory-mapped files may change frequently.

We use this key intuition to propose agile paging that combines the best of shadow and nested paging by:

1. using shadow paging for fast TLB misses for the parts of the guest page table that remain static, and

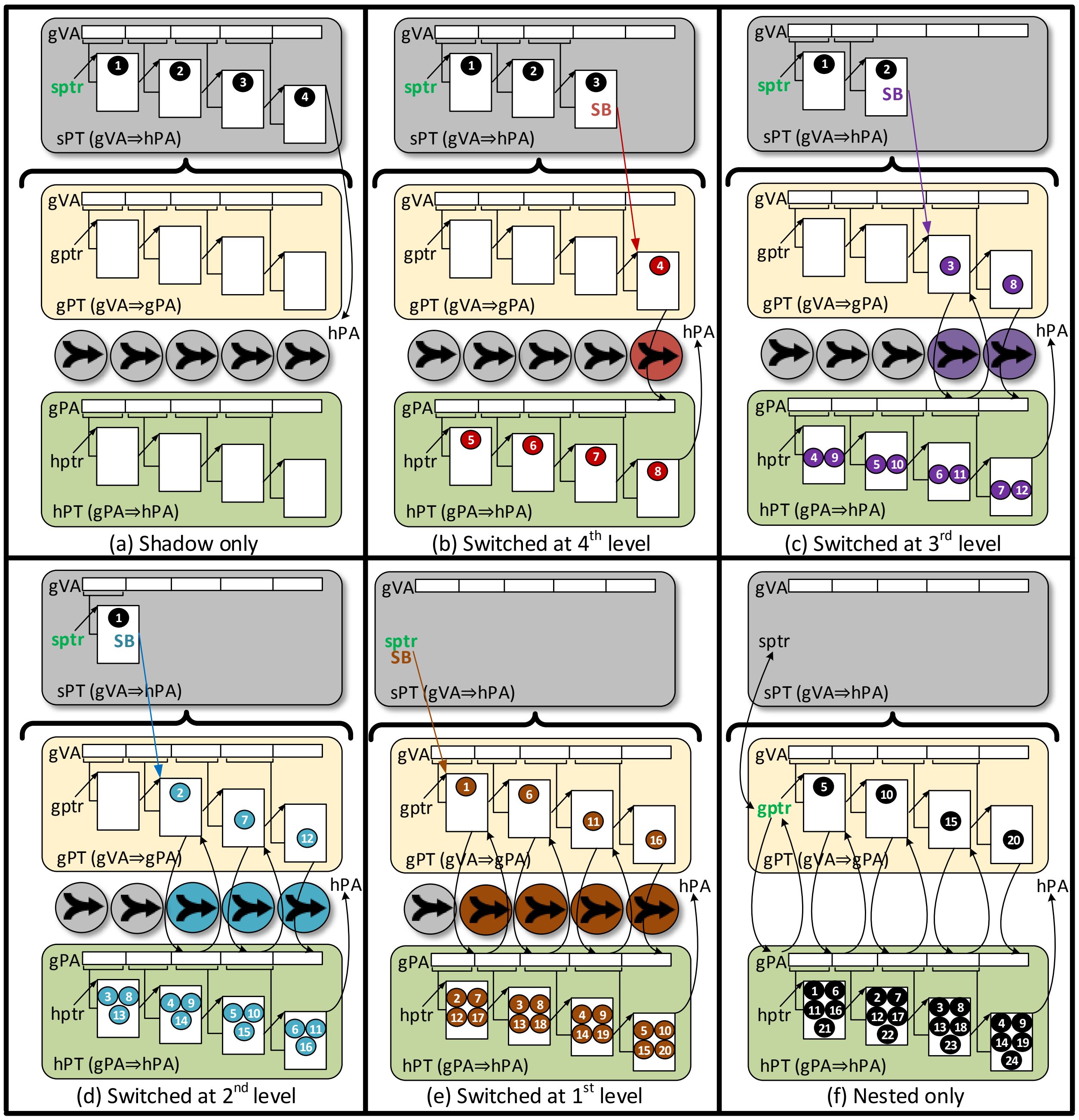
gVA
sptr
sPT (gVA⇒hPA)
gptr
gPT (gVA⇒gPA)
hPA
gPA
hptr
hPT (gPA⇒hPA)
SB
(a) Shadow only
(b) Switched at 4th level
(c) Switched at 3rd level
(d) Switched at 2nd level
(e) Switched at 1st level
(f) Nested only

Figure 5.2: Different degrees of nesting with agile paging in increasing order of page walk latency. Starting point for each is marked bold and green.

2. using nested paging for fast in-place updates for the parts of the guest page tables that dynamically change.

We refer to these two memory virtualization techniques as *constituent techniques* for

```
agile_walk(gVA, gptr, hptr, sptr)
if sptr==gptr then
    return nested_walk(gVA, gptr, hptr);
else
    nested = sPT.switching_bit;
    hPA = sptr;
    for (i=0; i⩽MAX_LEVELS; i++) do
        if nested then
            hPA = nested_PT_access(hPA + index(gVA,i), hptr);
        else
            hPA = host_PT_access(hPA + index(gVA,i));
            PTE = *(hPA + index(gVA,i));
            //Switching to nested mode
            if PTE.switching_bit then
                nested = true;
        end
    end
end
return hPA;
```

Figure 5.3: Pseudocode of hardware page walk state machine for agile paging. Note that agile paging requires modest switching support (shown in red) in addition to the state machines of nested and shadow paging shown in Sections 2.2.1 & 2.2.2 respectively.

the rest of the chapter. We show that agile paging performs better than its constituent techniques and supports features of conventional paging on both guest OS and VMM.

In the following subsections, we describe the hardware mechanism which will enable us to use both constituent techniques at the same time for a guest process and discuss policies that are used by the VMM to reduce overheads.

5.2.1 Mechanism: Hardware Support

Agile paging allows using the constituent techniques for the same guest process—even on a single address translation—with modest hardware support to switch between the two. Agile paging has three architectural page table pointers in hardware: one each for shadow, guest, and host page tables. If agile paging is enabled, virtualized page walk starts in shadow paging and then switches, in the same page walk, to nested paging if required. To allow fine grain switching from shadow paging to nested paging on any entry at any level of guest page table, the shadow page table needs to logically support a new *switching bit* per page table entry. This notifies the hardware page table walker to switch from shadow to nested mode. We choose not to support the switching in the other direction (nested to shadow mode) since the updates to the page tables are mostly confined to the lower levels of the page tables. When the switching bit is set in a shadow page table entry, the shadow page table holds the hPA (pointer) of the next *guest page table* level.

There are different degrees of nesting for virtualized address translation with agile paging: full shadow paging, full nested paging, and four degrees of nesting where translation starts in shadow mode and switches to nested mode at any level of the page table. These are shown in increasing order of page walk latency in Figure 5.2. The hardware page-walk state machine uses a bit to switch between the paging mechanisms as shown in Figure 5.3. A modest change needed to switch between the two techniques; the rest of the state machine is already present to support the constituent techniques. This change is shown in red in Figure 5.3.

Page Walk Caches: Modern processors have hardware page walk caches (PWCs) to reduce the number of memory accesses required for a page walk by caching the most-recently-used partial translations. For example, Intel processors use three partial translation tables inside PWCs: one table each to help skip the top one, two, or three levels of the page table [24, 30]. With shadow paging, PWCs store the hPA as a pointer to the next level of the shadow page table and thus skip accessing a few levels of the the shadow page table. With nested paging, PWCs store the hPA as a pointer to the next level of the guest page table, and skip accessing some of the levels of guest page table as well their corresponding host page table accesses. With agile paging, PWCs can be used to store partial translations for up to three levels of the guest page table without any restrictions on which mode any of the levels may be in. The PWCs will store an hPA for the partial translation with a single bit to denote whether the hPA points to shadow or guest page table so that an agile page walk can continue in the correct mode. While we extended Intel's PWCs with agile paging, the approach supports other designs as well.

5.2.2 Mechanism: VMM Support

Like shadow paging, the VMM for agile paging manages three page tables: guest, shadow, and host. Agile paging's page table management is closely related to that of shadow paging, but there are subtle differences.

Guest Page Table (gVA⇒gPA): As with shadow paging, the guest page table is created and modified by the guest OS for every guest process. The VMM, though, controls access to the guest page table by marking them read-only. Any attempt by the guest OS to change the guest page table will lead to a VMM intervention, which then updates the shadow page table to maintain coherence [17].

With agile paging, we leverage the support for marking guest page tables read-only with one subtle change. The VMM marks as read-only just the parts of the guest page table covered by the partial shadow page table. The rest of the guest page table (considered under nested mode) has full read-write access. Section 5.2.3 describes policies to choose what part of page table is under which mode. For example, KVM [72] allows the leaf level of a guest page table to be writable temporarily, called an unsynced shadow page, allowing multiple updates without intervening VMtraps. We extend that support to make other levels of the guest page table writable in our prototype.

Shadow Page Table (gVA⇒hPA): As with shadow paging, for all guest processes with agile paging enabled, a shadow page table is created and maintained by the VMM. The VMM creates this page table by merging the guest page table with the host page table so that any guest virtual address is directly converted to a host physical address. The VMM creates and keeps the shadow page table consistent [17].

However, with agile paging, the shadow page table is partial and cannot translate all gVAs fully. The shadow page table entry at each switching point holds the hPA of the next level of guest page table with the switching bit set (as shown in Figure 5.2). This enables hardware to perform the page walk correctly with agile paging using both techniques.

Host Page Table (gPA⇒hPA): As with shadow paging, the VMM manages the host page table to map from gPA to hPA for each virtual machine. VMM merges this page table with the guest page table to create a shadow page table. The VMM must update the shadow page table on any changes to the host page table. The host page table is only updated by the VMM and during that update the shadow page table is kept consistent by invalidating affected entries.

For standard shadow paging, the host page table is never referenced by hardware, and hence VMM can use other data structures instead of the architectural page-table format. However, with agile paging, the processor will walk the host page table for addresses using nested mode (at any level), and hence the VMM must build and maintain a complete host page table for each guest virtual machine as in nested paging.

Accessed and Dirty Bits: As with shadow paging, accessed and dirty bits are handled by the VMM and kept consistent between shadow page table and guest page table. On the first reference to a page, the VMM sets the accessed bit in the guest PTE and in the newly created shadow PTE. The write-enable bit is not propagated to the new shadow PTEs from the guest PTE. This ensures that the first write to the page will cause a protection fault, which causes a VMtrap that checks the guest PTE for write enable bit. At this point, the dirty bit is set in both the guest and shadow PTEs, and the shadow PTE is updated to enable write access to the page. If the guest OS resets any of these bits, the writes to guest page table are intercepted by the VMM which invalidates (or updates) the corresponding shadow PTEs.

With agile paging, we use the same technique for pages completely translated by shadow mode. Pages that end in nested mode instead use the hardware page walker, available for nested paging, to update guest page table accessed and dirty bits. We describe an optional hardware optimization in Section 5.3 that improves handling of accessed and dirty bits by eliminating costly VMtraps involved with shadow mode.

Context-Switches: Context switches within the guest OS are fast with nested paging, since guest OS is allowed to write to guest page table register. But with shadow paging, the VMM must intervene on context switches to determine the shadow page table pointer for the next process.

With agile paging, the context switching follows the mechanism used by shadow paging for all processes. The guest OS writes to the guest page table register, which triggers a trap to the VMM. The VMM finds the corresponding shadow page table and sets it in the shadow page table register. Hence, the cost of a context switch with agile paging is similar to shadow paging. We describe an optional hardware optimization in Section 5.3 that improves context switches in a guest OS by eliminating costly VMtraps involved in shadow paging.

To summarize, the changes to the hardware and VMM to support agile paging is incremental, but they result in a powerful, efficient and robust mechanism. The design is applicable to architectures that support nested page tables (e.g., x86-64 and ARM) and any hypervisor can use this architectural support. The hypervisor modifications are modest if they support both shadow and nested paging (e.g., KVM [72], Xen [23], VMware [100] and HyperV [9]).

5.2.3 Policies: What degree of nesting to use?

Agile paging provides a mechanism for virtualized address translation that starts in shadow mode and switches at some level of the guest page table to nested mode. The purpose of a policy is to determine *whether* to switch from shadow to nested mode for a single virtualized address translation and at *which level* of the guest page table the switch should be performed.

The ideal policy would determine that the page table entries are changing rapidly enough and the cost of corresponding updates to the shadow page table outweighs the benefit of faster TLB misses in shadow mode and thus the translation should use nested mode. The policy would quickly detect the dynamically changing parts of the guest page table and switch them to nested mode while keeping the rest of the static parts of the guest page table under shadow mode. Note that programs with very few TLB misses should use nested paging for the whole address space, as shadow mode has no benefit.

To achieve the above goal, a policy will move some parts of the guest page table from shadow to nested mode and vice-versa. We assume that the guest process starts in full shadow mode. We propose one static policy to move parts from shadow to nested mode and two online policies to move parts back from nested to shadow mode.

Shadow⇒Nested mode: Detecting dynamically changing parts of a guest page table is convenient when these parts are in shadow mode. These parts are marked read-only, thus any attempt to change an entry requires a VMM intervention (Section 5.2.2). Agile paging uses this to track the dynamic parts of the guest page table in the VMM and move those parts to nested mode.

To design a policy, we observe that updates to a page in page table are bimodal at a time interval of 1 second: only one update or many updates (e.g., 10, 50 or 500) within a second. Similar observations were made by Linux-KVM developers and used it to guide unsyncing a page of shadow page table [4]. For agile paging, if two writes to any level of the page table are detected by the VMM in a fixed time interval, then that level and all levels below it are moved to nested mode. This policy provides a small threshold like the one used in branch predictors for switching modes.

Nested⇒Shadow mode: The second, more complex, part of the policy is to detect when the workload changes behavior and stops changing the guest page table dynamically. This requires the switching parts of the guest page table back from nested to shadow mode to minimize TLB miss latency.

Our first simple online policy moves all the parts of the guest page table from nested back to shadow mode at fixed time interval and then use the above policy to move dynamic parts of the guest page table back to nested mode. While this policy is simple, it can lead to high overheads if the parts of the guest page table oscillate between the two modes.

A second more complex but effective policy uses dirty bits on the nested parts of the guest page table to detect changes to the guest page table itself. Under this policy, at the start of a fixed time interval, the VMM clears the dirty bits on the host page table entries mapping the pages of the guest page table. At the end of the interval, the VMM scans the host page table to which guest page table pages have dirty bits, which indicates the dynamic parts of the guest page table under nested mode. The non-dynamic parts of the guest page table (pages which did not have the dirty bit set) are switched back to shadow mode. The parent level of the guest page table is converted to shadow mode before converting child levels.

Short-Lived or Small Processes: Nested paging performs well for short-lived processes and for processes that have a very small memory-footprint since they do not run long enough to amortize the cost of constructing a shadow page table or do not suffer from TLB misses [29]. With agile paging, an administrative policy can be made to start the process in nested mode (no use of shadow mode) and turn on shadow mode after a small time interval (e.g., 1 sec) if TLB miss overhead is sufficiently large. The VMM can measure the TLB miss overhead with help of hardware performance counters and perform the switch to use agile paging.

To summarize, with our proposed policies, the VMM detects changes to the page tables and intelligently makes a decision to switch modes to reduce overheads.

5.3 Hardware Optimizations

Shadow paging was developed as a software-only technique to virtualize memory before there was hardware support. We propose two optional hardware optimizations that can further reduce the number of VMtraps associated with shadow paging and agile paging's shadow mode.

Handling Accessed and Dirty Bits: Agile paging requires costly VMtraps to keep accessed and dirty bits synchronized for regions of guest page table under shadow mode. Unlike shadow paging, in agile paging, the hardware has access to all three page tables (guest, host, shadow). As a result, we propose to extend hardware to set the accessed/dirty bit in all three page tables rather than just in the shadow. The extra page walk required to perform the write of accessed/dirty bits requires a full nested walk (up to 24 memory accesses) and will be faster than a long VMtrap. In addition, recent Intel Broadwell processors introduced two hardware page walkers per-core to help handle multiple outstanding TLB misses and writing accessed/dirty bits. Similar hardware for page walkers can be leveraged to perform writes to all page tables in parallel. Thus, in the worst case, on first write to a page will cost a two-level TLB miss to update the dirty bit.

Context-Switches: With every guest process context switch, the guest OS writes to the guest page table register, but is not allowed to set the shadow page table register since it does not have knowledge about the shadow page table. This results in costly VMtraps on context switches, which can degrade performance for workloads that do so frequently. In order to avoid these VMtraps, we propose adding a small 4-8 entry hardware cache to hold shadow page table pointers and their corresponding guest page table pointer, similar to how a TLB holds physical page numbers corresponding to virtual frame numbers. This cache can be filled and managed by the VMM (with help of new virtualization extensions) and accessed by the hardware on a context switch. So, if the guest OS writes to guest page table pointer register, hardware quickly checks this cache to see if there exists a shadow page table pointer corresponding to that guest process. On a hit, the hardware sets the shadow page table register without a VMtrap.

5.4 Paging Benefits

Agile paging is flexible and supports all features of conventional paging. We next describe how three important paging features that are supported with agile paging.

Large Page Support: Current processors support larger page sizes (2MB and 1GB pages in x86-64) by reducing the levels of the page tables and mapping larger regions of aligned contiguous virtual memory to aligned contiguous physical memory. Larger page sizes reduce the number of TLB misses for workloads with large working sets. Larger page sizes are supported at either or both stages for virtualized address translation with both shadow and nested paging; when large pages are used only in one stage of translation (e.g., guest only), they are in effect broken into smaller pages for entry into the TLB.

With agile paging, larger page sizes are supported using their current implementation in shadow paging and nested paging. The shadow page table, guest page table and host page table support larger page sizes as they all are multi-level page tables and can reduce their depth. Thus, agile paging supports larger page sizes using the same mechanisms and policies described in Section 5.2.

Content-Based Page Sharing: Content-based page sharing is a technique used by both the guest OS and the VMM to save memory for many workloads. The guest OS or VMM scans memory to find pages with identical content. When such pages are found, the guest OS or VMM reclaims all but one copy and maps all the copies using copy-on-write [100]. This mechanism shares pages within a process, between two guest processes and even between two virtual machines. Reclamation by the guest OS requires changes to the guest page table (and shadow page table if applicable) whereas reclamation by the VMM requires changes to the host page table (and shadow page table if applicable).

As copy-on-write must update the page table, agile paging will naturally detect these changes and move parts of the page table to nested mode to reduce overheads. The overhead of copy-on-write is very high with shadow paging and will benefit from nested mode provided by agile paging.

Memory pressure: When free memory is scarce, a guest OS will frequently scan and clear the referenced bits of page tables looking for pages to reclaim (e.g., clock algorithm). With shadow paging, this scanning causes VMtraps, which increases overhead on an already stressed system. With agile paging, though, the VMM detects the page-table writes to clear referenced bits and converts leaf-level page tables to nested mode to avoid the VMtraps.

To summarize, existing page-based mechanisms blend naturally with our proposed technique. Agile paging has the *agility* to adapt to changing environments and is powerful

Processor	Dual-socket Intel Xeon E5-2430 (Sandy Bridge), 6 cores/socket 2 threads/core, 2.2GHz
Memory	96 GB DDR3 1066MHz
OS	Linux kernel version 3.12.13
VMM	QEMU (with KVM) version 1.6.2, 24vCPUs
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

Table 5.3: System configurations and per-core TLB hierarchy.

to reduce overheads of memory virtualization.

5.5 Methodology

To evaluate our proposal, we emulate our proposed hardware with Linux and prototype our software in KVM.

State of the art: We compare against six configurations: base native (B), nested paging (N) and shadow paging (S), each with two page sizes 4KB and 2MB. The prior work, SHSP [101] performs similarly to the best of shadow and nested paging, so we do not evaluate it separately.

We run workloads to completion on real hardware as described in Table 6.5. We use Linux's *perf utility* [10] to measure (i) total execution cycles for all six configurations: base native (E_B), nested paging (E_N) and shadow paging (E_S) for both page sizes, (ii) number of TLB misses ($M_{B/N/S}$) (iii) cycles spent on TLB misses ($T_{B/N/S}$) (iv) cycles spent in the hypervisor ($H_{B/N/S}$) (v) number of VMtraps ($V_{B/N/S}$) for each page size. For the virtualized setting, we use the same page size for both levels of address translation since they only

Ideal execution time (from base native)	$E^{ideal} = E^{2M} - T^{2M}$
Overhead of page walks (for both 4K and 2M)	$PW_{B/N/S} = [E_{B/N/S} - E^{ideal} - H_{B/N/S}]/E^{ideal}$
Overhead of VMM (for both 4K and 2M)	$VMM_{B/N/S} = H_{B/N/S}/E^{ideal}$
Avg. cycles per TLB miss (for both 4K and 2M)	$C_{B/N/S} = T_{B/N/S}/M_{B/N/S}$
Overhead of page walk (A) (for both 4K and 2M)	$PW^{A} = [C_N * \sum_{i=2}^{4}(F_{Ni}) + C_S * (1 - \sum_{i=1}^{4} F_{Ni})$ $+(C_N + C_S) * 0.5 * F_{N1}] * M_B$
Overhead of VMM (A) (for both 4K and 2M)	$VMM^{A} = O_S - \sum_{i=1}^{n}(F_{Vi} * C_{Ei})$

Table 5.4: Performance model based on performance counters and BadgerTrap.

reduce TLB misses when both have same page size. For transparently using 2MB and 4KB page sizes, we turn on transparent huge page support in Linux [15]. Note that the effects of various caching techniques like caching of PTEs in data caches [65], page walk caches [24, 30], Intel EPT caches and nested TLBs [28, 29] are already included in the performance measurement since they are part of the base commodity processor.

Linux does not use 1GB pages transparently and instead requires applications to explicitly allocate 1GB pages. Thus, we did not include that configuration. Agile paging supports 1GB page size (Section 5.4) and has the potential to reduce overheads of page walks with 1GB page size.

We calculate the overhead of page walks and that of VMM interventions and report those for each of the six configurations based on the performance model described in Table 5.4.

Agile Paging: We use a novel two-step approach to report improvements for agile paging. We first generate a trace of page table updates from KVM and then use that trace with BadgerTrap [52] (Chapter 3) to calculate performance using a linear model. Next we describe our novel two-step methodology.

Suite	Description	Workload	Memory
SPEC 2006	compute and memory intensive single-threaded workloads	astar	350 MB
		gcc	885 MB
		mcf	1.7 GB
PARSEC	Shared-memory multi-threaded workloads	canneal	780 MB
		dedup	1.4 GB
BioBench	Bioinformatics single-threaded workloads	tigr	610 MB
Big Memory	Generation, compression and search of graphs	Graph500	73 GB
	In-memory key-value cache	Memcached	75 GB

Table 5.5: Workload description and memory footprint.

Step 1: The goal of this step is to create a list of dynamically changing gVAs to classify under nested mode and calculate the fraction of VMtraps (F_{Vi}) that agile paging eliminates with reason "i". This emulates the optional hardware optimizations for agile paging as well. The workload is run to completion with shadow paging on real hardware (described in Table 6.5) with an instrumented VMM to create a per-vCPU trace of all updates to the guest and host page table that lead to shadow page table updates. We use the trace-cmd tool [14] with KVM, modified to print extra trace information, for creation of the trace in this step. We process the trace to find which areas of the page tables are changing by looking at the reasons for VMtraps. We record the gVAs being dynamically changed due to changes on any level of the page table. This helps us create four lists of gVAs under nested mode corresponding to their switching level of page table. The lists of gVAs are considered under nested paging for step 2. This step also finds the fraction of VMM interventions that agile paging will reduce (F_{Vi}) from the trace since areas of the page table under nested paging are known. Note that we emulate our shadow-to-nested policy in an offline fashion when processing the trace.

Step 2: The goal of this step is to find the fraction of TLB misses that would be serviced under nested mode (F_{Ni}) for each level "i" the switch occurs. The workload is again run to completion, but this time with nested paging along with BadgerTrap [52]: a tool that converts all x86-64 TLB misses to a trap, which allows us to analyze TLB misses and classify TLB addresses, while enabling full-speed execution of instructions with TLB hits (Chapter 3). We instrument the TLB misses and classify them under shadow mode or nested mode at each switching level. We compare TLB miss addresses against the gVAs for nested mode to find the fraction of TLB misses serviced in nested mode at each switching level i (F_{Ni}). We conservatively assume that when a TLB miss is serviced in nested mode F_{N1} pays half the cost of a nested TLB miss beyond native and the rest F_{N2}, F_{N3} and F_{N4} pays full cost of nested paging. This assumption leads to higher overheads for agile paging than with real hardware.

Cost of VMtraps: We measure VMtrap latency as the cycles to complete the VMM intervention: the VMexit operation and return plus the work done by the VMM in response to the VMexit. The total cost varies and can be 1000s of cycles. We use LMbench [6] and microbenchmarks to measure the cost of the VMtrap for a context switch, page table update and page fault. We calculate reduction in cycles spent in the VMM, by subtracting the number of VMtraps of each type multiplied by its cost.

Performance Model: We use the fractions calculated using the above two step approach to develop a linear model to project performance for agile paging (A) with both page sizes (i) fraction of VMtraps reduced (F_{Vi}) and, (ii) fraction of TLB misses serviced under nested mode (F_{Ni}) for both page sizes. The linear model takes the performance of shadow paging, subtracts the cost of VMtraps avoided, but adds in the higher cost of nested TLB misses.

This linear performance model is similar to previous research [30, 53, 68, 83].

We use this new two step approach with the linear model to evaluate realistic workloads (in time and data footprint). We use a linear performance model to illuminate trends. Our two-phase approach includes the real system effects of page sharing, zapping of shadow entries, page walk cache, etc., and approximates the performance impact of these effects on agile paging.

Benchmarks: Our proposal is applicable to a wide variety of workloads from desktop to big-memory workloads. To evaluate our proposal, we select workloads with high TLB-miss overhead (more than 5MPKI) from SPEC [59], PARSEC [34], BioBench [22], and big-memory workloads [25] as summarized in Table 5.5.

5.6 Results

This section evaluates the cost of address translation and VMM interventions of agile paging, and shows that agile paging outperforms state-of-the-art techniques.

5.6.1 Performance analysis

Figure 5.4 shows the execution time overheads associated with page walk and VMM interventions for each workload in eight different configurations: base native paging (bars 4K:B and 2M:B), nested paging (bars 4K:N and 2M:N), shadow paging (bars 4K:S and 2M:S) and agile paging (bars 4K:A and 2M:A). Each bar is split into two segments. The bottom segment represents the overheads associated with page walks and the top dashed segment represents the overheads associated with VMM interventions.

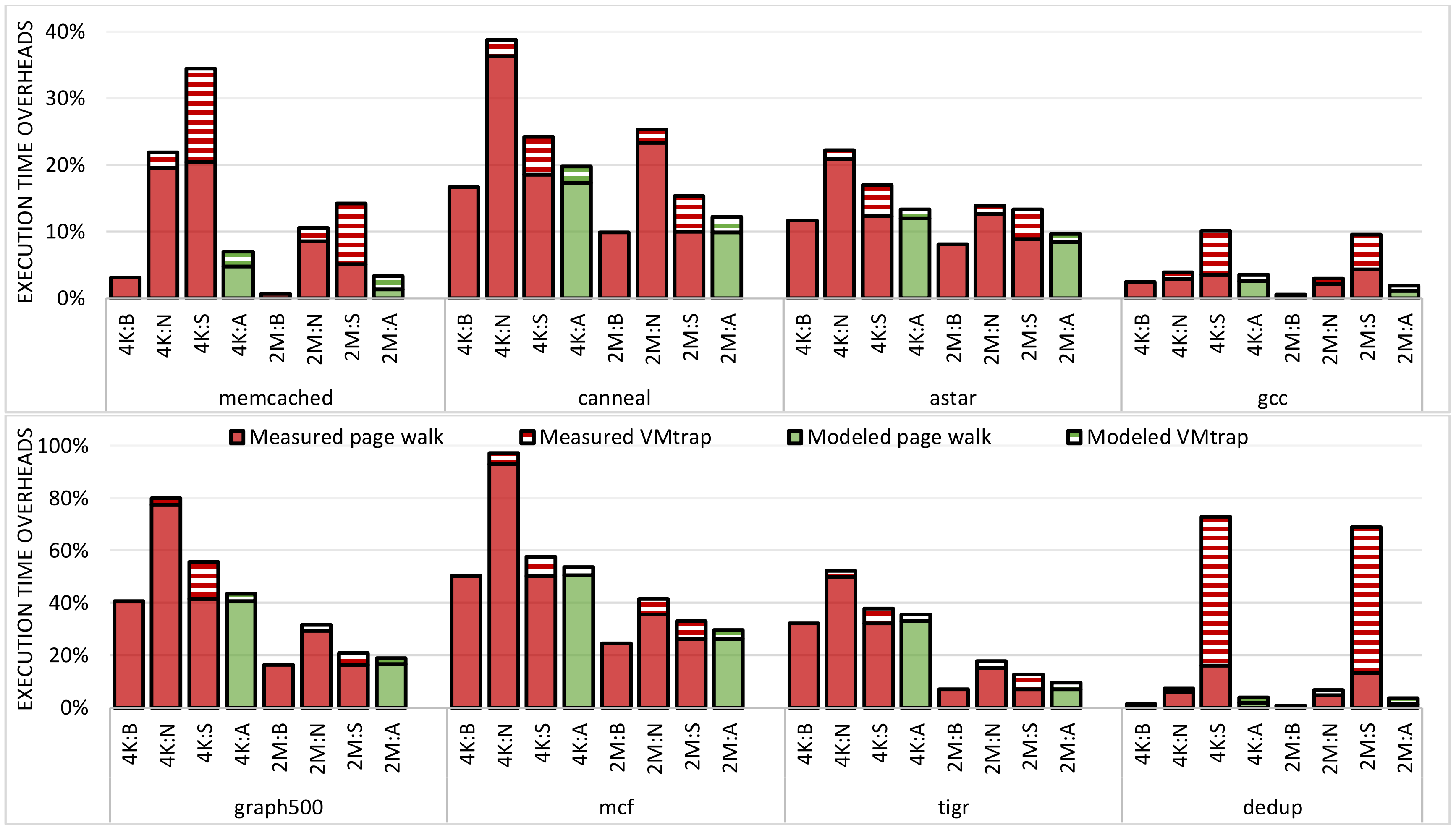

Figure 5.4: Execution time overheads due to page walks (bottom bar) and VMM interventions (top dashed bar) for all workloads.

Agile paging outperforms its constituent techniques for all workloads and improves performance by 12% over the best of nested and shadow paging on average. It performs less than *4% slower* than unvirtualized native at worst. On the other hand, nested and shadow paging perform 31% and 70% slower than unvirtualized native at worst. We find:

1. *For these workloads, the base native system with 4KB pages incurs high overheads.* For example, mcf, tigr and graph500 spend 50%, 32% and 41% respectively.

2. *With virtualization, the overheads increase drastically under nested paging.* For example, the overheads increase from 50% to 97% for mcf (bar 4K:B vs. bar 4K:N). Compared to native, translation overheads increase by 2.5$\times$ with nested paging using 4KB pages (geometric mean).

3. *Shadow paging has high cost due to VMM interventions for some workloads.* While shadow paging generally performs better than nested paging, it has high overheads for dedup, memcached and gcc. Dedup has 57% overhead spent in the VMM servicing page table updates. Compared to native, translation overheads increase by 3.1$\times$ with shadow paging using 4KB pages.

4. *Agile paging outperforms the best of shadow paging and nested paging for all workloads.* Agile paging achieves low cost of VMM interventions as well low latency TLB misses. Agile paging is only 2.3% slower than native on average (up to 3.7%). These results include the benefits of hardware optimizations.

5. *2MB large pages help reduce overheads of virtual memory. Agile paging helps reduce overheads further.* Large pages consistently improve performance with agile paging as compared to the 4KB page size.

Switch Level	Shadow	L4	L3	L2	L1	Nested	
Mem. accesses	4	8	12	16	20	24	Avg.
memcached	88.2%	4.5%	7.3%	0%	0%	0%	4.76
canneal	94.7%	4.6%	0.7%	0%	0%	0%	4.24
astar	92.3%	7.5%	0.2%	0%	0%	0%	4.32
gcc	81.6%	11.7%	6.7%	0%	0%	0%	5.00
graph500	99.8%	0.2%	0%	0%	0%	0%	4.01
mcf	99.1%	0.9%	0%	0%	0%	0%	4.04
tigr	88.3%	7.6%	3.1%	0%	0%	0%	4.51
dedup	91.4%	2.2%	6.4%	0%	0%	0%	4.60

Table 5.6: Percentage of TLB misses covered by each mode of agile paging while using 4KB pages assuming no page walk caches. Most of the TLB misses are served in shadow mode.

5.6.2 Insights into Performance of Agile Paging

We report the fraction of TLB misses covered by each mode of agile paging in Table 5.6 for 4KB pages. For this table alone, we assume no page walk caches. More than 80% of TLB misses are covered under complete shadow mode reducing TLB miss latency to 4 or 5 memory accesses. Thus, few of the pages suffering TLB misses also have frequent page-table updates. By converting the changing portion of the guest page table to nested mode, agile paging prevents most of the VMexits that makes shadow paging slower. We also note that most of the upper-levels of the page table remain static after initialization and hence use shadow mode. Overall, the average number of memory accesses for a TLB miss comes down from 24 to between 4-5 for all workloads.

5.6.3 Discussion: Selective Hardware Software Paging (SHSP)

SHSP seeks to select the best paging technique dynamically [101]. Their results for a 64-bit VM running SPEC workloads showed that SHSP achieves approximately the best of the two techniques. While SHSP, with no hardware support, improves on a single system-wide choice, it is limited by the high cost of each virtualization technique alone. Agile paging breaks that limitation and exceeds the best of shadow and nested paging by more than 12% on average, even with a pessimistic model. Moreover, Table 5.6 shows that only part of the address space needs to switch from one mode to the other.

5.7 Summary

We and others have found that the overheads of virtualizing memory can be high. This is true, in part, because currently guest processes must choose between (i) nesting paging with slow 2D page table walks or (ii) shadow paging wherein page table updates cause costly VMM interventions. Ideally, one would want to use nested paging for addresses and page table levels that change, and use shadow paging for addresses and page table levels that are relatively static.

Our proposal—*Agile Paging*—seeks the above ideal. With Agile Paging, a virtualized address translation usually starts in shadow mode and then switches to nested mode only if required to avoid VMM interventions. Agile paging requires a new hardware mechanism to switch modes during a page walk in response to a page-table-entry bit set by the VMM. The VMM policy seeks to invoke the mode change only on more dynamic page table entries.

We emulate the proposed hardware and prototype the software in Linux with KVM on x86-64. We find that, for our workloads, agile paging robustly performs better than the best of nested paging and shadow paging.

Chapter 6

EXPLOITING RANGE TRANSLATIONS WITH REDUNDANT MEMORY MAPPINGS

6.1 Introduction

Direct segments [25, 53] trades the flexibility of paging for performance which is good for some applications, but not all. In addition, we learned from agile paging, that we can preserve flexibility of paging and still reduce overheads of virtual memory. So, we find a robust solution to reduce overheads of virtual memory in a native systems.

Chapter 1 shows that the performance of paging is suffering due to stagnant TLB sizes, whereas modern memory capacities continue to grow, causing the problem of *limited TLB reach*. Recent studies show that modern workloads can experience execution-time overheads of up to 50% due to page table walks [25, 30, 69]. 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 [25, 50].

Previous research has focused on solving this problem by improving the efficiency of paging in the following three ways.

	Transparent to application	Kernel support	Hardware support	# of entries	Maximum reach per entry	Application domain	No size-alignment restrictions
Multipage Mappings [81, 82, 96]	✓	✗	✓	512	32 KB to 16 MB	any	✗
Transparent Huge Pages [15, 77]	✓	✓	✓	32	2 MB	any	✗
libhugetlbfs [1]	✗	✓	✓	4	1 GB	big memory	✗
Direct segments [25]	✗	✓	✓	1	unlimited	big memory	✓
Redundant Memory Mappings	✓	✓	✓	N	unlimited	any	✓

Table 6.1: Comparison of Redundant Memory Mappings with previous approaches for reducing virtual memory overhead.

1. Multipage mappings use one TLB entry to map multiple pages (e.g., 8-16 pages per entry) [81, 82, 96]. Mapping multiple pages per entry increases TLB reach by a 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 [15] and libhugetlbfs [1]) 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 and standard paging for the remaining virtual address space [25, 53]. 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 mechanism that is transparent to applications and improves translation performance 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 ranges 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.

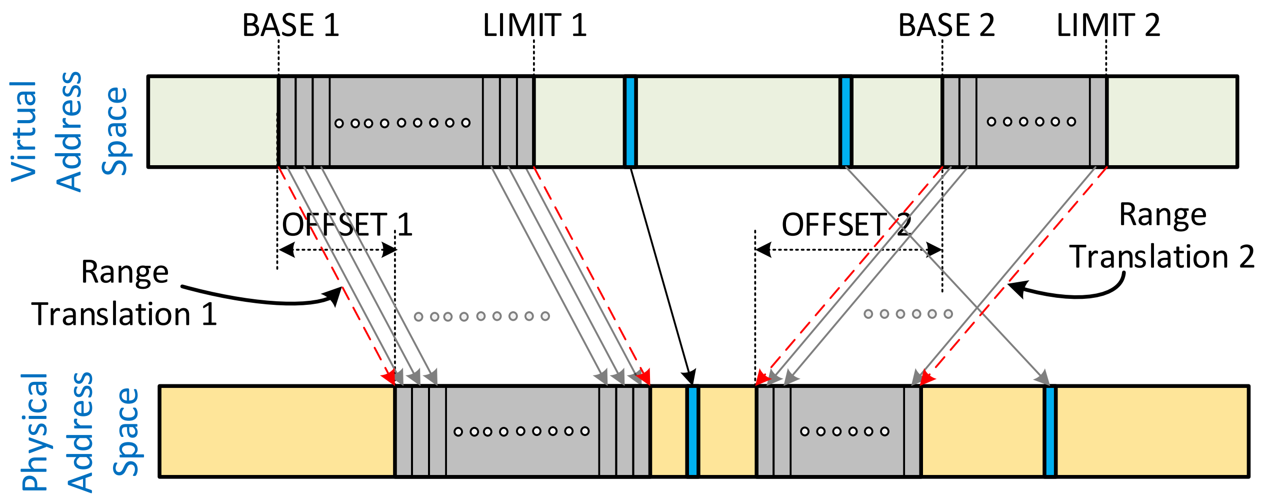

Figure 6.1: Range translation: an efficient representation of contiguous virtual pages mapped to contiguous physical pages.

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 6.1 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 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 change 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 [15]. 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 Section 6.2's Table 6.2).

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 [53] (Chapter 3)—the same methodology as in prior TLB studies [25, 30, 53]. 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 for our big-memory workloads.

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, and operating system modifications that detect and manage ranges.

	Huge 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 6.2: Total translation entries mapping the application's memory with: (i) Transparent Huge Pages of 4 KB and 2 MB pages [15] 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.

- 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.

This proposal was created in collaboration with Vasileios Karakostas with both of us as primary authors and was orginally published in the 42nd ACM/IEEE Symposium on Computer Architecture (ISCA-42) 2015 [68]. A shorter version of this proposal was selected for and published in IEEE Micro Special Issue: Micro's Top Picks from Architecture Conferences 2016 [55].

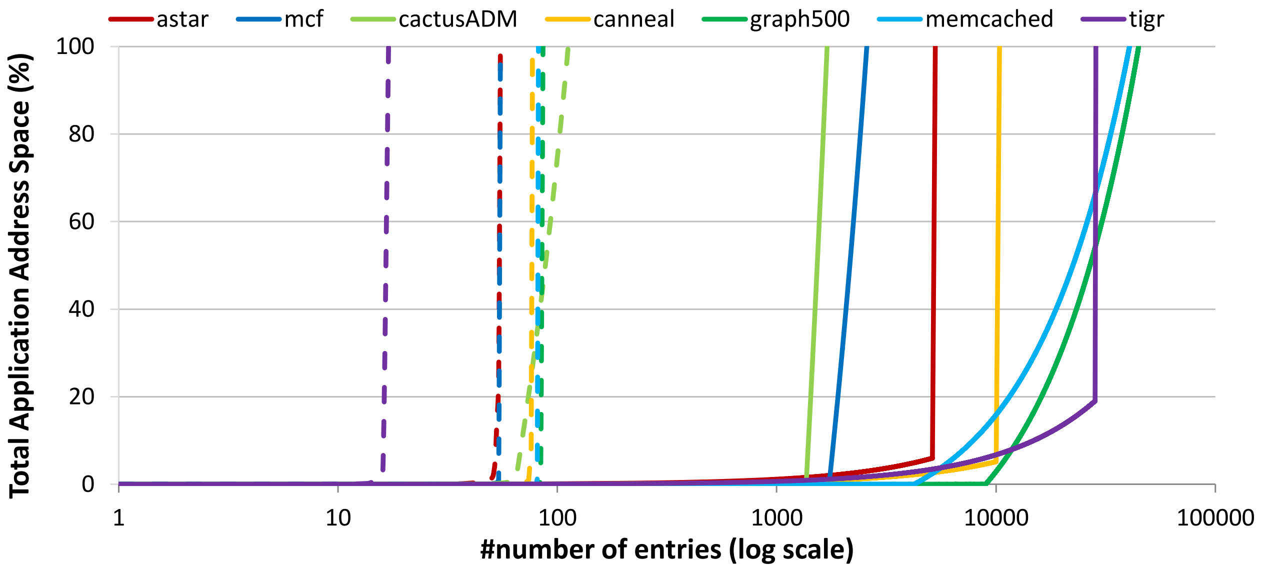

Figure 6.2: Cumulative distribution function of the application's memory (percentage) that N translation entries map with pages (solid) and with optimal ranges (dashed), for seven representative applications. Ranges map all applications' memory with one to four orders of magnitude fewer entries than pages.

6.2 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 6.6 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 6.2 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 fewer 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. Figure 6.2 plots the number of pages and contiguous virtual page ranges required to map all of an application's address space for seven representative workloads. Hence, a modest number of ranges have the potential to efficiently perform address translation for the majority of virtual memory addresses—orders of magnitude less than with regular or even huge PTEs.

6.2.1 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. To translate a virtual range address to physical address, the hardware adds the virtual address to the OFFSET of the corresponding range. Figure 6.3 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.

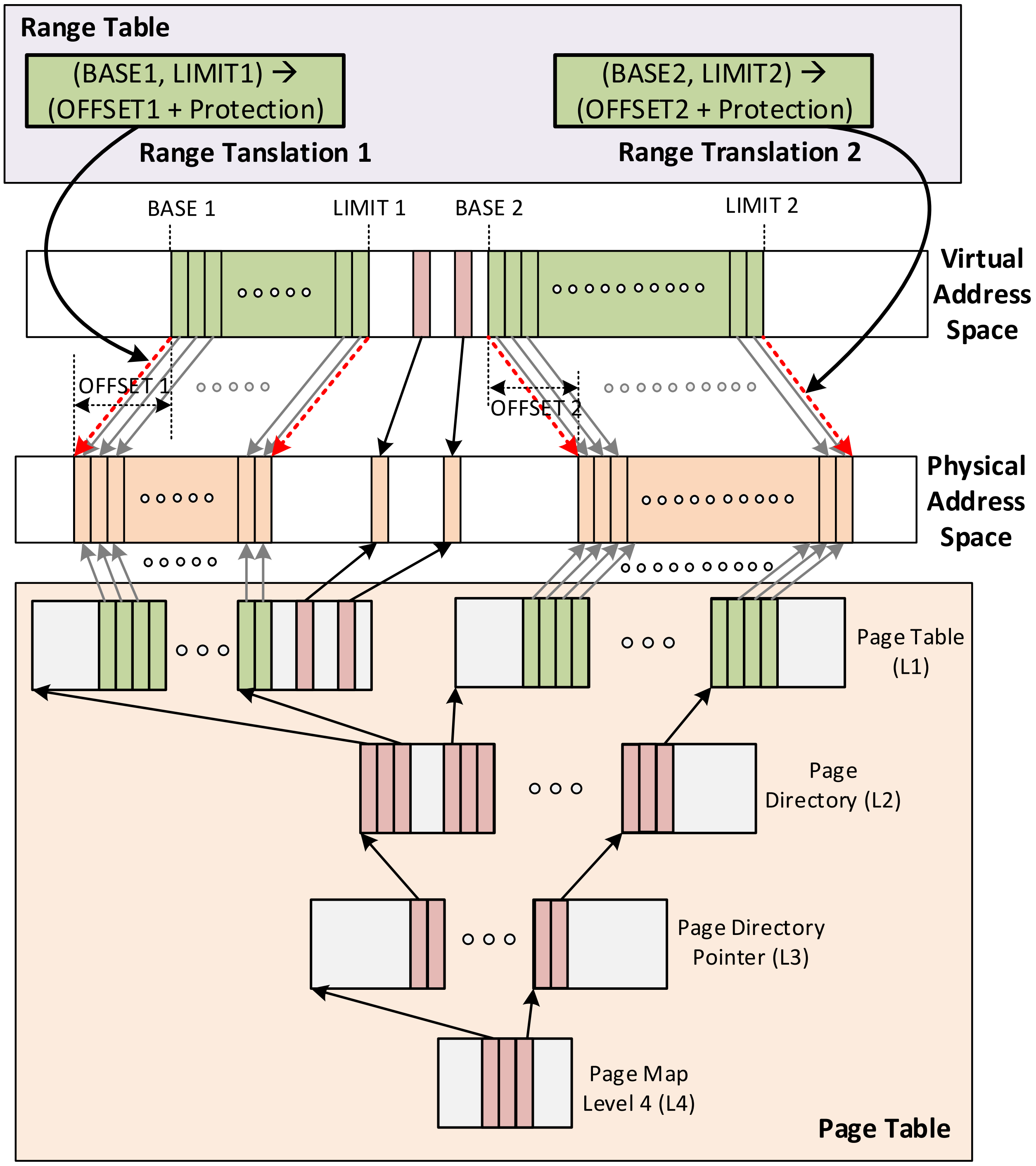

Figure 6.3: Redundant Memory Mappings design. The application's memory space is represented *redundantly* by both pages and range translations.

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

Table 6.3: Overview of Redundant Memory Mapping

We introduce three novel components to manage ranges: (i) *range TLB*s, (ii) *range tables*, and (iii) *eager paging* allocation. Table 6.3 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.

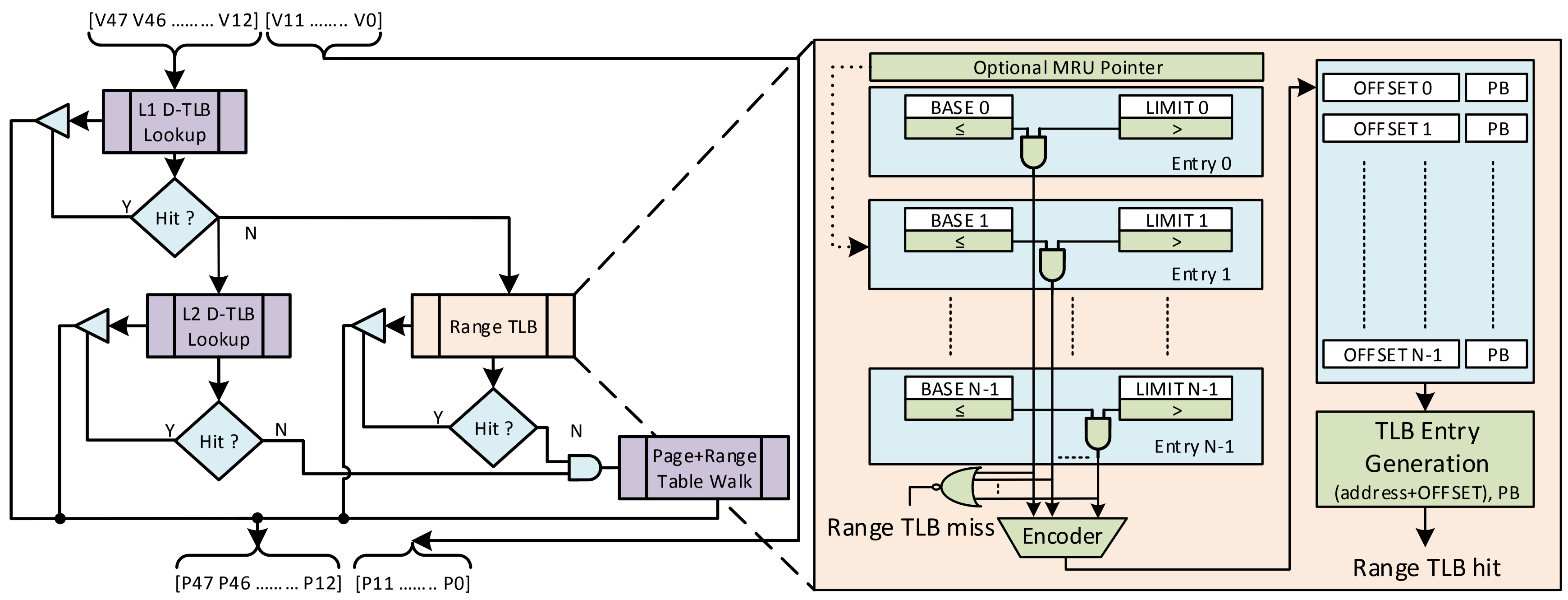

Figure 6.4: RMM hardware support consists primarily of a range TLB that is accessed in parallel with the last-level page TLB.

6.3 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. Below, we describe RMM as an extension to the x86-64 architecture, but the design applies to other architectures as well.

6.3.1 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 6.4 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 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.

Figure 6.4 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 $\text{BASE}_i \leqslant \text{virtual page number} < \text{LIMIT}_i$ 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. It 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 6.3.3 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 [65]. Unlike a page TLB, the range TLB is similar to N fully-associative copies of direct segment's base/limit/offset logic [25] or a simplified version of the range cache [99]: 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 [71]. Our results in Section 6.7 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 can be extended to the other translation levels closer to the processor (e.g., in parallel to the L1 TLB); we leave such analysis for future work.

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.

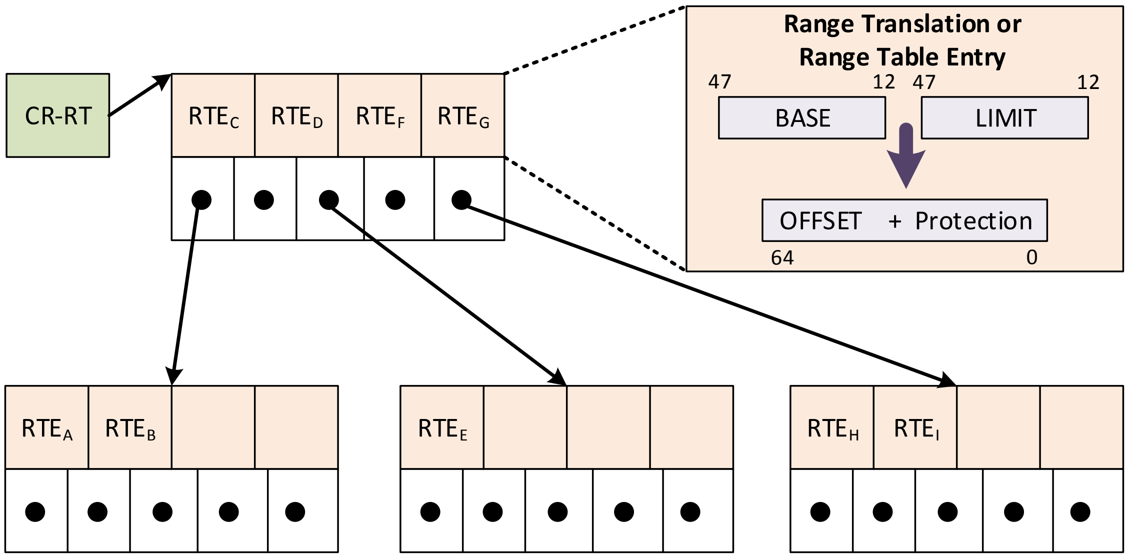

Figure 6.5: The range table stores the range translations for a process in memory. The OS manages the range table entries based on the applications memory management operations.

6.3.2 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 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 6.5 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 6.7). 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 explain next.

6.3.3 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 [36].

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.

6.4 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.

6.4.1 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.

6.4.2 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 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 $k = 1, 2, ...$ 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

```
compute the memory fragmentation;
if $memory\ fragmentation \leqslant threshold$ then
    // use eager paging;
    while $number\ of\ pages > 0$ do
        for $(i = MAX\_ORDER-1; i \geqslant 0; i–)$ do
            if $freelist[i] \geqslant 0$ and $2^i \leqslant number\ of\ pages$ 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
```

Figure 6.6: 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*.

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 [16]. However, one motivation for demand paging was to limit unnecessary swapping in multiprogrammed workloads, which modern large memories make less common [25]. We find that 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.

Algorithm. Figure 6.6 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 [81, 82].

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

6.5 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 a 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.

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 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 [42]. In this case, the OS could perform full compaction [25, 82], or partial compaction with techniques adapted from garbage collection [37, 42].

6.6 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

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 6.4: Workload description and memory footprint.

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

	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 6.5: System configurations and per-core TLB hierarchy.

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 6.3 with Intel Sandy Bridge core shown in Table 6.5. 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 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 [52] (Chapter 3). 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 this same approach [25, 30, 53].

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%.

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 6.5) with realistic input sets until completion and collect processor and TLB statistics using hardware performance counters. We use the Linux *perf* utility [10] 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 [25, 53] given in Table 6.6.

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 [22], Parsec [34] and big-memory workloads [25] as summarized in Table 6.4. We execute each application sequentially on a single test machine without rebooting between experiments.

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 6.6: Performance model based on hardware performance counters and BadgerTrap.

6.7 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 6.5) 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. We implement the Clustered TLB approach [81] of Pham et al., configured with 512 fully-associative entries. Each entry indexes up to an 8-page cluster, shown best by Clustered TLB [81]. 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 [25].

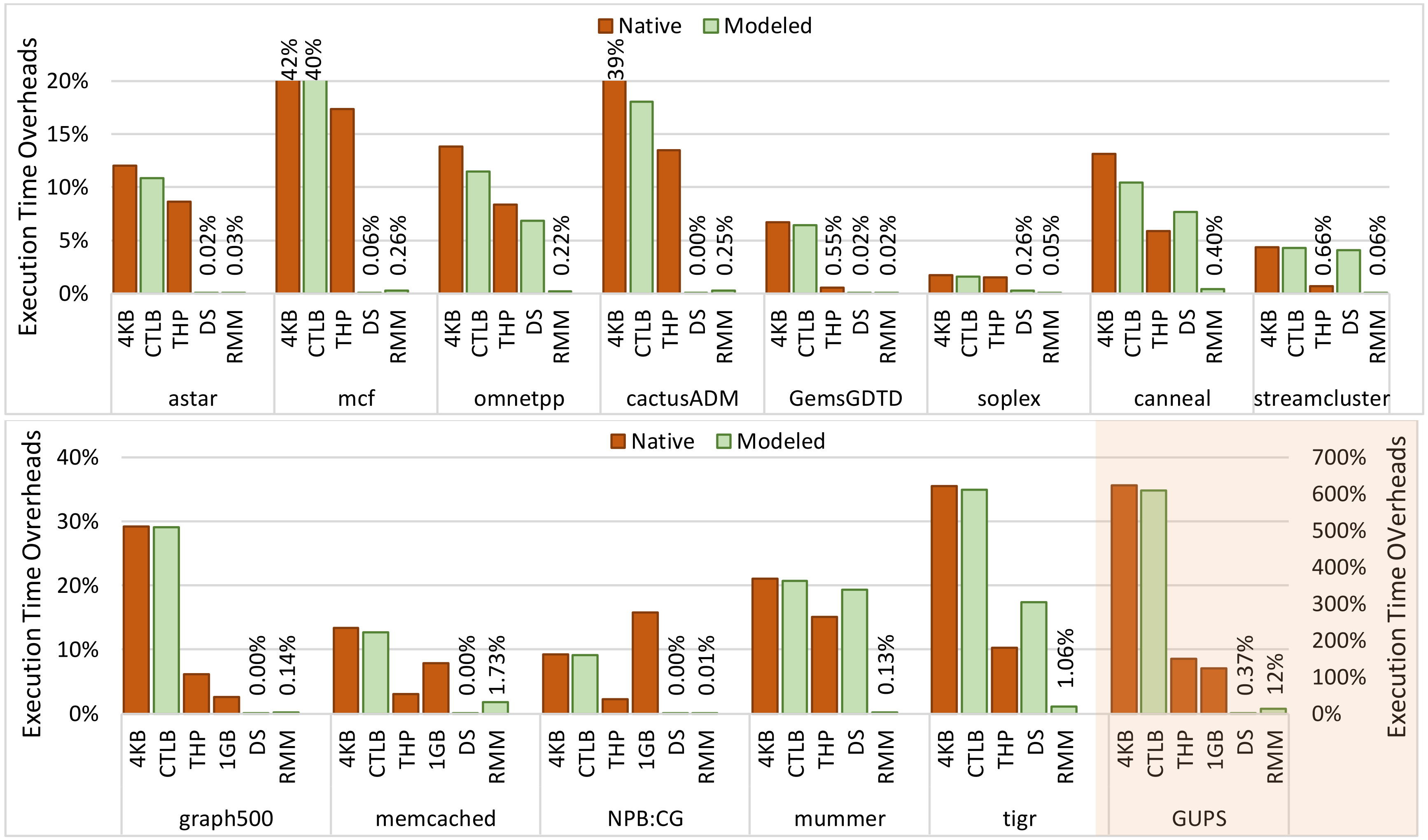

Figure 6.7: Execution time overheads due to page-walks for SPEC 2006 and PARSEC (top) big-memory and BioBench (bottom) workloads. GUPS uses the right y-axis and thus shaded separately. 1GB pages are only applicable to big-memory workloads.

6.7.1 Performance analysis

Figure 6.7 shows the overhead spent in page-walks for RMM compared to other techniques. The 4 KB, 2 MB Transparent Huge Pages (THP) [15] and 1 GB [1] 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 [81] results. The DS bars show direct segments [25] 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 to 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.

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. 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.

6.7.2 Range TLB sensitivity analysis

To achieve high performance, the range TLB must be large enough to satisfy most L1 TLB misses. Figure 6.8 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

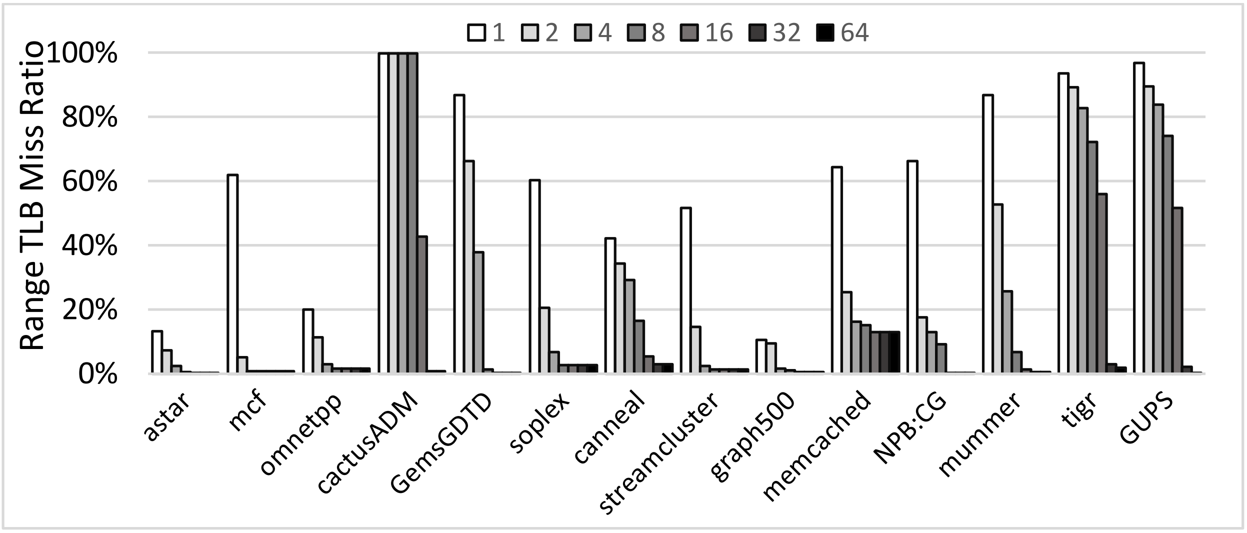

Figure 6.8: Range TLB miss ratio as a function of the number of range TLB entries.

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.

6.7.3 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 6.7 shows the effect of eager paging on the number and size of range, and on time and memory overheads, compared to default demand paging. Default demand paging includes forming THPs, which we translate to ranges.

	Demand Paging					Eager Paging						
			range size in 4 KB pages					range size in 4 KB pages			% time	% memory
Benchmark	# ranges	% memory	median	average	max	# ranges	% memory	median	average	max	overhead	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 6.7: Impact of eager paging on ranges, time, and memory compared to demand paging (with Transparent Huge Pages).

The first two sections of Table 6.7 (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.

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 6.7 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 [47, 91].

Eager paging anticipates that the application will use the requested memory regions and may thus increase the memory footprint. The last column of Table 6.7 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 6.5.

6.7.4 Energy

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

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 [84]). 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 6.5), 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.*[1] 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.

6.8 Summary

We propose Redundant Memory Mappings, a novel and robust translation mechanism, that improves performance by 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.

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

Chapter 7

SUMMARY AND FUTURE WORK

This chapter summarizes the thesis and mentions a few future research directions suggested by to this dissertation.

7.1 Summary

Virtual memory is a crucial abstraction in modern computer systems. However, modern workloads are experiencing execution time overheads of up to 50% due to page-based virtual memory. This problem is mainly caused by two opposing trends:

1. Physical memory is growing exponentially cheaper and bigger and modern workloads want to store ever increasing large data sets in memory and orders of maginitude more slowly.

2. Translation Lookaside Buffer (TLB) size has grown slowly than memory, because the TLB is on the processor's critical path to access memory.

We expect this mismatch between memory size and TLB size (i) to keep growing, (ii) to become worse with newer memory technologies, which promise petabytes of physical memory, and (iii) to increase the overheads of paging due to costly TLB misses. Our key observation is many features of traditional page-based virtual memory are underutilized in native systems and in virtual machines. This observation suggested opportunities to reduce the cost of virtualizing memory by enabling alternative mechanisms of address translation without significantly losing benefits of paging. This dissertation exploited these observations and provided fast address translation for a wide variety of workloads. This dissertation research aimed to achieve near-zero overheads of virtual memory for both native and virtualized systems primarily through three contributions.

First, we observed that the overheads of page-based virtual memory can increase drastically with virtual machines. Hardware performs a nested two-level translation to translate a guest virtual address. This additional cost makes virtualization less attractive for many workloads. In MICRO'14 [53] (Chapter 4), we proposed new hardware building on direct segments [25] with three new virtualized modes that significantly improveed virtualized address translation. The new hardware bypassed both or either levels of paging for most address translations using direct segments and brought down the dimensionality of page walk from 2D to a 0D (see Figure 7.1). This mechanism preserved properties of paging when necessary and provided fast translation by bypassing paging where unnecessary. This work was selected as one of 24 most significant papers of 2014 and received an honorable mention in IEEE MICRO for its novelty and long term impact (IEEE MICRO Top Picks 2015). This work also inspired the creation and release of a Linux based tool called BadgerTrap to analyze and instrument TLB misses [52], improving the methodology for analyzing and optimizingvirtual memory system design.

Virtualized direct segments bypassed widely used hardware support—nested paging—but ignored a less often used software technique—shadow paging. Moreover, virtualized direct segments required extensive hardware, VMM, and operating system support. We exploited the fact that nested and shadow paging—the state-of-the-art techniques—provide

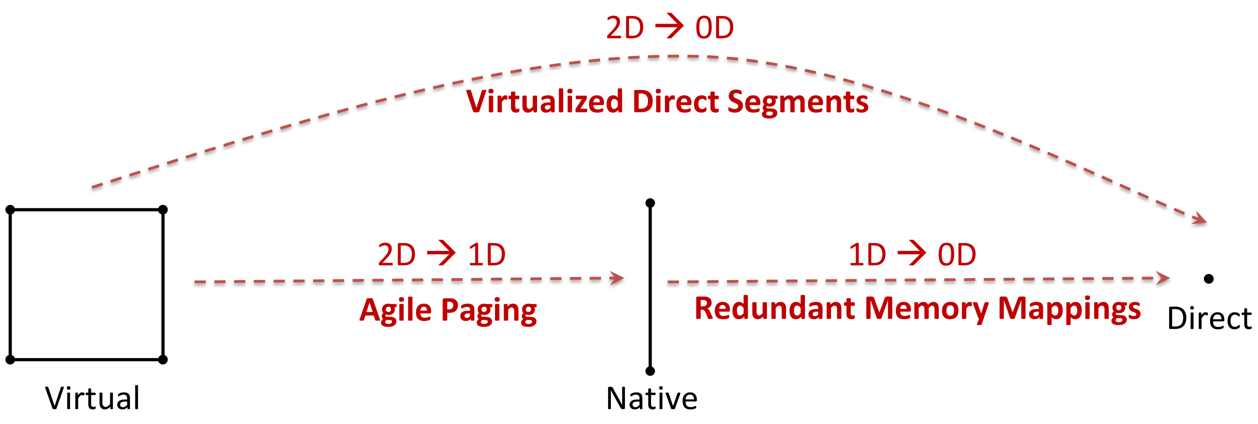

Figure 7.1: Effect of different proposals in this thesis on dimensionality of page walk.

different tradeoffs. With nested paging, TLB misses are costly due to nested two-level page walk, but it perform fast direct updates to page tables. Whereas with shadow paging, TLB misses have fewer memory acceses since hypervisor creates a single-level shadow page table, but requires costly hypervisor intervention on page table updates. For ISCA'16 [54] (Chapter 5), we proposed agile paging, which combines both the techniques. A virtualized page walk starts in shadow mode and then switches to nested mode for address ranges where frequent updates would cause costly hypervisor interventions. This mechanism reduced the cost of page walks, as many TLB misses are handled partially or entirely in shadow mode and helped in reducing the dimensionality of page walk from 2D to 1D (see Figure 7.1). In addition, this technique preserved all benefits of paging and robustly exceeded the performance of both the state-of-the-art techniques.

Direct segments [25] traded the flexibility of paging for performance, which is good for some applications, but not all. For ISCA'15 [68] (Chapter 6),we proposed range translations which exploit the virtual memory contiguity in modern workloads to perform address translation much more efficiently than paging. Inspired by direct segments, a range translation is a mapping between contiguous virtual pages mapped to contiguous physical pages of arbitrary size with uniform protection bits. We implemented range translations in the Redundant Memory Mappings (RMM) architecture. RMM employed hardware/software co-design to map the entire virtual address space with standard paging and redundantly mapped ranges with range translations. Since range translations were backed by page mappings, the OS may flexibly choose between using range translations or not, thus retaining the benefits of paging for fine-grain memory management when necessary. This work was selected as one of 12 most significant papers of 2015 and will published in IEEE MICRO Top Picks 2016 for its novelty and long term impact [55]. This proposal help reduce a 1D page walk to a 0D page walk for TLB misses that hit in Range TLB (see Figure 7.1).

7.2 Future Work

Over the past several years, a paradigm shift is occurring in microprocessor design industry. In the era of energy efficiency, specialization and heterogeneity is a driving design principle. On the computing side, we see a rise of various different resources like graphics processing units (GPUs) and other accelerators (database, crypto, etc.) being closely integrated with our general purpose processors. On the memory and storage side, we are seeing a rise in non-volatile memory (NVM) technologies alongside of traditional DRAM playing a big role and promises to increase memory capacity to petabytes with fast access times. This increase in heterogeneity opens up a plethora of multidisciplinary research opportunities spanning across computer architecture, operating systems, databases, graphics, programming languages, and applications. Inspired from this dissertation, interesting areas for future research are:

Efficient address translation with NVMs: Virtual memory in its current form is likely infeasible for mapping petabytes of NVM integrated with traditional DRAM. New memory technologies requires rethinking of how to integrate them in our memory hierarchy and how to virtualize them while delivering fast access time. Some of my previous experience with RMM and agile paging can be leveraged to create a virtual memory system for next generation of NVM-based systems.

Efficient support for nested virtualization: Recently, there has been an increasing interest in supporting nested virtualization—a VMM can run various VMMs which can run their own VMs [27, 102]. This approach comes with its own set of benefits:

(i) public cloud providers can give consumers full control of their VMs by making them run on their own favorite VMM,

(ii) increase security and integrity of VMs on public clouds even when the VMM provided by service provider is compromised [108],

(iii) enable new features used by VMMs which the service providers are slow to adopt (like VM migration [27]), and

(iv) help debug, build and benchmark new VMMs. Vendors like VMware have been supporting nested virtualization by using software techniques [20].

Hardware advancement such as VMCS shadowing are being included in recent processors (e.g., Intel Haswell) to perform efficient nested virtualization [43]. Unfortunately, these benefits of nested virtualization come at the cost of increasing the memory virtualization overheads because of the additional levels of abstraction. This new interest in nested virtualization motivates us to think about providing a scalable solution for memory virtualization in presence of more levels of virtualization.

NUMA-aware data placement and address translation: Designing large multi-socket servers comes with a tradeoff. On one hand, physically non-contiguous NUMA-aware data placement is necessary to achieve maximum bandwidth utilization. But on the other hand, virtual and physical contiguity is required for fast translation especially with larger memories [68]. This tradeoff can cause a workload to perform poorly in a large NUMA system. We would like to explore possibilities on improving efficiency of workloads on large NUMA machines by leveraging the concepts such as Redundant Memory Mappings [68].

Efficient virtualization of GPUs and other Accelerators: Many accelerators are virtualized very poorly by using I/O memory management unit (IOMMU). By using this interface, the accelerators are used as slow devices rather than tightly integrated co-processing accelerators. We would like to explore mechanisms to improve IOMMU support and provide hardware support to virtualize these accelerators to make them achieve their true potential as high performance accelerators rather than being treated as slow devices.

Memory consistency and persistency: NVM technologies have very different properties than traditional DRAM. Current memory consistency models are not sufficient to handle ordering of operation to persistent NVM memory efficiently and provide crash consistency guarantees. To cope with this deficiency, memory persistency was proposed to enhance memory consistency rules in presence of persistent memory [80]. These persistency orderings have implications on our cache hierarchy since caches have been tuned to provide efficient access to traditional DRAM memory. Designing an efficient persistent memory interface so that programs can easily use it while integrating effortlessly into the memory hierarchy for fast access is an important and open problem.

In conclusion, with the end of Moore's law in sight, we must consider more creative cross-layer solutions—a philosophy that this thesis embraced. In our opinion, this philosophy will provide higher performance and energy efficiency over the next decade—the golden age for hardware/software co-design of computer systems.

BIBLIOGRAPHY

[1] Huge Pages Part 1 (Introduction). http://lwn.net/Articles/374424/.

[2] Intel 8086. https://en.wikipedia.org/wiki/Intel_8086.

[3] KSM - KVM. http://www.linux-kvm.org/page/KSM.

[4] KVM MMU Virtualization. https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_guangrong.pdf.

[5] Linux Virtio Balloon Driver. http://lxr.freeelectrons.com/source/drivers/virtio/virtio_balloon.c.

[6] LMbench: Tools for Performance Analysis. http://www.bitmover.com/lmbench/.

[7] Memory compaction. http://lwn.net/Articles/368869/.

[8] Memory Hotplug. https://www.kernel.org/doc/Documentation/memory-hotplug.txt.

[9] Microsoft Virtualization: Hyper-V. https://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx.

[10] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[11] Predictive Failure Analysis (PFA). https://msdn.microsoft.com/en-us/library/windows/hardware/ff559459(v=vs.85).aspx.

[12] SPCS001: Intel Next-Generation Haswell Microarchitecture. http://blog.scottlowe.org/2012/09/11/spcs001-intel-next-generation-haswell-microarchitecture/.

[13] TCMalloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[14] trace-cmd: A front-end for Ftrace. https://lwn.net/Articles/410200/.

[15] Transparent huge pages in 2.6.38. http://lwn.net/Articles/423584/.

[16] IBM Systems Reference Library: IBM OS Linkage Editor and Loader. http://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/360/os/R21.0_Mar72/GC28-6538-9_OS_Linkage_Editor_and_Loader_Release_21_Jan72.pdf, 1972.

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

[18] Advance Micro Devices. *Software Optimization Guide for AMD Family 15h Processors*. Number 47414. 2014.

[19] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The evolution of an x86 virtual machine monitor. *SIGOPS Oper. Syst. Rev.*, 44(4):3–18, December 2010.

[20] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

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

[22] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce 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, 2005*, pages 2–9, 2005.

[23] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

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

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

[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, Washington, DC, USA, 2012. IEEE Computer Society.

[27] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles

project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

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

[29] Nikhil Bhatia. Performance evaluation of Intel EPT hardware assist. `http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf`, 2009.

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

[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, Washington, DC, USA, 2011. IEEE Computer Society.

[32] 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, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative tlb for chip multiprocessors. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 359–370, New York, NY, USA, 2010. ACM.

[34] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[35] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[36] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. 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, New York, NY, USA, 1989. ACM.

[37] 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 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[38] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and RH Taheri. Methodology for performance analysis of VMware vSphere under Tier-1 applications. *VMware Technical Journal*, 2(1), 2013.

[39] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, New York, NY, USA, 1997. ACM.

[40] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. Improving virtualization in the presence of software managed

translation lookaside buffers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 120–129, New York, NY, USA, 2013. ACM.

[41] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 128–137, New York, NY, USA, 1994. ACM.

[42] Nachshon Cohen and Erez Petrank. Limitations of partial compaction: Towards practical bounds. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 309–320, New York, NY, USA, 2013. ACM.

[43] Intel Corporation. 4th Generation Intel Core vPro Processors with Intel VMCS Shadowing. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf`.

[44] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D, June 2016. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[45] Intel Corporation. Intel Itanium Architecture Software Developer's Manual Revision 2.3 Volumes 1-4. `http://www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html`.

[46] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in multics. *Commun. ACM*, 11(5):306–312, May 1968.

[47] 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, London, UK, UK, 2000. Springer-Verlag.

[48] Yu Du, Miao Zhou, Bruce R. Childers, Danie Mossé, and Rami Melhem. Supporting superpages in non-contiguous physical memory. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 223–234, Feb 2015.

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

[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. *SIGPLAN Not.*, 47(4):37–48, March 2012.

[51] Narayanan Ganapathy and Curt Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’98, pages 8–8, Berkeley, CA, USA, 1998. USENIX Association.

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

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

[54] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Exceeding the best of nested and shadow paging. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, ISCA ’16, New York, NY, USA, 2016. ACM.

[55] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, 36(3):118–126, May 2016.

[56] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(9):34–45, September 1974.

[57] 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, New York, NY, USA, 2012. ACM.

[58] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, Mar 2014.

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

[60] Giang Hoang, Chang Bae, John Lange, Lide Zhang, Peter Dinda, and Russ Joseph. A case for alternative nested paging models for virtualized systems. *IEEE Computer Architecture Letters*, 9(1):17–20, Jan 2010.

[61] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122, New York, NY, USA, 2012. ACM.

[62] Sun Microsystems Inc. UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007. `http://www.oracle.com/technetwork/systems/opensparc/t2-14-ust2-uasuppl-draft-hp-ext-1537761.html`, Sept 2007.

[63] Intel. An Introduction to SR-IOV Technology. `http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html`, 2011.

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

[65] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual, April 2012.

[66] Bruce Jacob and Trevor Mudge. Uniprocessor virtual memory without tlbs. *IEEE Trans. Comput.*, 50(5):482–499, May 2001.

[67] 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, Washington, DC, USA, 2002. IEEE Computer Society.

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

[69] 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*, pages 1–12, 2014.

[70] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, April 1962.

[71] 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, 2007.

[72] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007.

[73] William Lonergan and Paul King. Design of the b 5000 system. *IEEE Ann. Hist. Comput.*, 9(1):16–22, January 1987.

[74] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

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

[76] MIPS Technologies, Incorporated. MIPS32 Architecture for Programmers Volume iii: The MIPS Privileged Resource Architecture, 2001.

[77] 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 implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, pages 89–104, Berkeley, CA, USA, 2002. USENIX Association.

[78] D. L. Osisek, K. M. Jackson, and P. H. Gum. Esa/390 interpretive-execution architecture, foundation for vm/esa. *IBM Syst. J.*, 30(1):34–51, February 1991.

[79] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. Prediction-based superpage-friendly tlb designs. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 210–222, Feb 2015.

[80] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[81] 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. IEEE, 2014.

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

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

[84] Stephen Phillips. M7: Next Generation SPARC. In *Hot Chips: A Symposium on High Performance Chips*, 2014.

[85] 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, 2013.

[86] Nathan E. Rosenblum, Gregory Cooksey, and Barton P. Miller. Virtual machine-provided context sensitive page mappings. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 81–90, New York, NY, USA, 2008. ACM.

[87] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[88] 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, New York, NY, USA, 2000. ACM.

[89] Andreas Sembrant, Erik Hagersten, and David Black-Schaffer. The direct-to-data (d2d) cache: Navigating the cache hierarchy with a single lookup. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 133–144, Piscataway, NJ, USA, 2014. IEEE Press.

[90] Andreas Sembrant, Erik Hagersten, and David Black-Shaffer. Tlc: A tag-less cache for reducing dynamic first level cache energy. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 49–61, New York, NY, USA, 2013. ACM.

[91] 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, New York, NY, USA, 1993. ACM.

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

[93] 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, March 2012.

[94] Indira Subramanian, Cliff Mather, Kurt Peterson, and Balakrishna Raghunath. Implementation of multiple pagesize support in hp-ux. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 9–9, Berkeley, CA, USA, 1998. USENIX Association.

[95] Mark Swanson, Leigh Stoller, and John Carter. Increasing tlb reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 204–213, Washington, DC, USA, 1998. IEEE Computer Society.

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

[97] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 415–424, New York, NY, USA, 1992. ACM.

[98] Dong Tang, Peter Carruthers, Zuheir Totari, and Michael W. Shapiro. Assessment of the effect of memory page retirement on system ras against hardware faults. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 365–370, Washington, DC, USA, 2006. IEEE Computer Society.

[99] 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, Washington, DC, USA, 2008. IEEE Computer Society.

[100] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, pages 181–194, New York, NY, USA, 2002. ACM.

[101] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Selective hardware/software memory virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 217–226, New York, NY, USA, 2011. ACM.

[102] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 113–126, New York, NY, USA, 2012. ACM.

[103] 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, New York, NY, USA, 2002. ACM.

[104] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM.

[105] David A. Wood, Susan J. Eggers, Garth Gibson, Mark D. Hill, Joan M. Pendleton, Scott A. Ritchie, George S. Taylor, Randy H. Katz, and David A. Patterson. An in-

cache address translation mechanism. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 358–365, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[106] Hongil Yoon and Gurindar S. Sohi. Revisiting virtual l1 caches: A practical design using dynamic synonym remapping. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 212–224, March 2016.

[107] Kazutomo Yoshii, Kamil Iskra, Harish Naik, Pete Beckmanm, and P. Chris Broekema. Characterizing the performance of "big memory" on blue gene linux. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 65–72, Washington, DC, USA, 2009. IEEE Computer Society.

[108] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM.