

RICE UNIVERSITY

**Exploiting Address Space Contiguity to Accelerate TLB  
Miss Handling**

by

**Thomas W. Barr**

A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

**Master of Science**

APPROVED, THESIS COMMITTEE:

---

Alan L. Cox, Chair  
Associate Professor of Computer Science  
and Electrical and Computer Engineering

---

Scott Rixner  
Associate Professor of Computer Science  
and Electrical and Computer Engineering

---

Peter Varman  
Professor of Electrical and Computer  
Engineering and Computer Science

Houston, Texas

April, 2010

## ABSTRACT

### Exploiting Address Space Contiguity to Accelerate TLB Miss Handling

by

Thomas W. Barr

The traditional CPU-bound applications of the past have been replaced by multiple concurrent data-driven applications that use lots of memory. These applications, including databases and virtualization, put high stress on the virtual memory system which can have up to a 50% performance overhead for some applications. Virtualization compounds this problem, where the overhead can be upwards of 90%. While much research has been done on reducing the number of TLB misses, they can not be eliminated entirely. This thesis examines three techniques for reducing the cost of TLB miss handling. We test each against real-world workloads and find that the techniques that exploit coarse-grained locality in virtual address use and contiguity found in page tables show the best performance.

The first technique reduces the overhead of multi-level page tables, such as those used in x86-64, with a dedicated *MMU cache*. We show that the most effective MMU caches are *translation caches*, which store partial translations and allow the page walk hardware to skip one or more levels of the page table. In recent years, both AMD and Intel processors have implemented MMU caches. However, their implementations are quite different and represent distinct points in the design space. This thesis introduces three new MMU cache structures that round out the design space and directly compares the effectiveness of all five organizations. This comparison shows that two of the newly introduced structures, both of which are translation cache variants, are better than existing structures in many situations.

Secondly, this thesis examines the relative effectiveness of different page table organizations. Generally speaking, earlier studies concluded that organizations based on hashing, such as the inverted page table, outperformed organizations based upon radix trees for supporting large virtual address spaces. However, these studies did not take into account the possibility of caching page table entries from the higher levels of the radix tree. This work shows that any of the five MMU cache structures will reduce radix tree page table DRAM accesses far below an inverted page table.

Finally, we present a novel device, the SpecTLB, that is able to exploit alignment in the mapping from virtual address to physical address to interpolate translations without any memory accesses at all. Operating system support for automatic page size selection leaves many small pages aligned within large page “reservations”. While large pages improve TLB coverage, they limit the control the operating system has over memory allocation and protection. Our device allows the latency penalty of small pages to be avoided while maintaining fine-grained allocation and protection.

## **Acknowledgments**

There is a long list of people that deserve great thanks for enabling the work in this thesis to even be possible. It would be impossible to list them all here, but I would be remiss if I didn't try. First, I thank my committee members for their constant and detailed feedback and their unwavering motivation. The diversity of viewpoints granted to me from working with Dr. Alan L. Cox, Dr. Scott Rixner and Dr. Peter Varman have been entirely invaluable. The ability to compile a single work from so many areas of computing is a rare honor, one that would never have been possible without their help.

Secondly, I must thank all the people who indirectly helped me complete this work. While a project such as this is never easy, they have made the past year of my life a delight. The constant, unquestioning support of my closest family and friends allowed me to bounce back from the many failures along the way. I dedicate my successes to you.

---

# Contents

---

Abstract	ii
List of Illustrations	viii
List of Tables	x
<b>1 Introduction</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Contributions . . . . .	4
1.3 Organization . . . . .	6
<b>2 x86 Address Translation</b>	<b>7</b>
2.1 Page Walk Details . . . . .	7
<b>3 Related Work</b>	<b>12</b>
3.1 Caching . . . . .	12
3.2 Alternate page table formats . . . . .	13
3.3 Reducing TLB miss frequency . . . . .	14
<b>4 MMU Caching</b>	<b>16</b>
4.1 Introduction . . . . .	16
4.2 Caching page walks . . . . .	18
4.2.1 Page table caches . . . . .	19
4.2.2 Translation caches . . . . .	21
4.2.3 Translation-Path Cache (TPC) . . . . .	24
4.2.4 Design space summary . . . . .	25
4.3 Design comparison . . . . .	25
4.3.1 Indexing . . . . .	26

4.3.2	Partitioning . . . . .	27
4.3.3	Coverage . . . . .	29
4.4	Implementation . . . . .	30
4.4.1	CAM Bypass . . . . .	31
4.4.2	VI-LRU shift register implementation . . . . .	34
4.5	Methodology . . . . .	36
4.5.1	Application Memory Traces . . . . .	36
4.5.2	Memory System Simulation . . . . .	37
4.5.3	Synthetic Application Memory Traces . . . . .	38
4.6	Cache design simulations . . . . .	38
4.6.1	TLB miss penalty . . . . .	39
4.6.2	Sizing considerations . . . . .	43
4.6.3	Replacement policy . . . . .	46
4.7	Virtualization . . . . .	47
4.8	Conclusions . . . . .	49
<b>5</b>	<b>Alternate page table formats</b>	<b>53</b>
5.1	Hashed page tables . . . . .	53
5.2	Translation Storage Buffers . . . . .	57
5.3	Conclusion . . . . .	58
<b>6</b>	<b>SpecTLB: Parallelizing TLB Miss Handling</b>	<b>59</b>
6.1	Background . . . . .	61
6.1.1	FreeBSD Reservation System . . . . .	61
6.2	Page table speculation . . . . .	62
6.2.1	Explicit page table marking . . . . .	63
6.2.2	Heuristic reservation detection . . . . .	65
6.2.3	Memory side-effects . . . . .	66
6.3	Methodology . . . . .	67

6.3.1	Platform simulator . . . . .	67
6.3.2	Benchmarks . . . . .	68
6.3.3	SpecTLB Simulation . . . . .	68
6.4	Simulation results . . . . .	69
6.4.1	TLB miss parallelization . . . . .	69
6.4.2	Overlap opportunity . . . . .	71
6.4.3	Power overhead . . . . .	71
6.4.4	Sizing considerations . . . . .	72
6.4.5	Replacement policy . . . . .	73
6.5	Discussion . . . . .	74
6.5.1	More aggressive reservation creation . . . . .	74
6.5.2	Very large page support for virtual machines . . . . .	75
6.5.3	Explicit marking . . . . .	76
6.6	Conclusions . . . . .	77
<b>7</b>	<b>Conclusions</b>	<b>81</b>
7.1	Future work . . . . .	84
<b>A</b>	<b>MMU Coverage Proofs</b>	<b>86</b>
A.1	Comparison of TPC and Unified Coverage . . . . .	87
A.2	Comparison of TPC and SPTC Coverage . . . . .	89
	<b>Bibliography</b>	<b>91</b>

---

## Illustrations

---

2.1	Decomposition of the x86-64 virtual address. . . . .	8
2.2	An example page walk for virtual address (0b9, 00c, 0ae, 0c2, 016). Each page table entry stores the physical page number for either the next lower level page table page (for L4, L3, and L2) or the data page (for L1). Only 12 bits of the 40-bit physical page number are shown in these figures for simplicity. . . . .	9
4.1	An example of the contents of a UPTC. . . . .	19
4.2	An example of the contents of a SPTC. . . . .	20
4.3	An example of the contents of a STC. . . . .	22
4.4	An example of the contents of a UTC. . . . .	22
4.5	An example of the contents of the TPC. . . . .	24
4.6	Fully associative cache design. . . . .	32
4.7	Dynamic bit logic circuit. . . . .	33
4.8	Dynamic bit logic circuit with bypass. . . . .	34
4.9	A shift register. . . . .	35
4.10	A VI-LRU shift register. . . . .	35
4.11	L3 page table hit rate for the 16GB database join. . . . .	44
4.12	L2 page table hit rate for Sweep3d . . . . .	45
4.13	L2 page table hit rate for CactusADM . . . . .	46
4.14	L3 page table hit rate for the 16GB database using VI-LRU replacement. . .	48
6.1	A page table containing a marked reservation. . . . .	64
6.2	An example of the contents of a SpecTLB. . . . .	64



6.3	Speculation rates for different sizes of SpecTLB. . . . .	72
6.4	Speculation rates for different replacement policies. . . . .	73

---

## Tables

---

4.1	General parameters for different cache designs. . . . .	30
4.2	Specific parameters for different cache designs on AMD64. . . . .	31
4.3	The frequency of TLB misses for each workload, shown as the number of instructions, memory accesses and DRAM accesses between TLB misses. These results are shown for three different L2 data cache sizes and the TLB configuration described in Section 4.5. . . . .	40
4.4	TLB memory access patterns for various benchmarks. . . . .	51
4.5	Effect of replacement policy on MMU cache miss rate. . . . .	52
5.1	L2 hits and DRAM accesses to the page table per walk for a radix tree page table for the 16GB database join benchmark. Results are shown for an uncached radix tree, a cached radix tree, and a half-full inverted page table with various numbers of clustered translations per tag. . . . .	55
6.1	SpecTLB simulation results for SPEC CPU2006. . . . .	79
6.2	SpecTLB simulation results for other benchmarks. . . . .	80

# CHAPTER 1

---

## Introduction

---

### 1.1 Introduction

Virtual memory is not a solved problem. While it is conceptually simple, mapping multiple large virtual address spaces to a single physical address space is a massive organizational challenge. There can be millions of pages resident in memory at any given time. Additionally, virtual address spaces are large and sparsely used. Even though a particular process may only use a few gigabytes of memory, this data may be spread over a many terabyte region of the virtual address space. To support this, page tables on x86-64 use a multi-level radix tree. This format saves space, but requires multiple accesses to the page table to translate a single address.

On modern systems, this *address translation* becomes the dominant performance overhead of virtual memory. The limited size of translation lookaside buffers (TLBs) means that misses are inevitable and the page table must be consulted to translate addresses. Recent work has shown that the impact of address translation on overall system performance ranges from 5-14% even for nominally sized applications in a non-virtualized environment [1].

As the application's memory footprint increases, it has a significantly larger impact on virtual memory performance, approaching 50% in some cases [2]. In this thesis, we show that databases can be particularly taxing on address translation hardware. A common oper-

ation in databases is the join, which can use hash tables of many gigabytes for in-memory databases. Such joins are common, and the performance of large joins is representative of overall database performance [3]. Modern scientific supercomputing workloads are working with fewer dense matrices and more with sparse and graph structures [4]. These workloads have less spatial locality in their access pattern and therefore will see a far higher TLB miss rate than traditional scientific applications. In the future, all of these applications are likely to use more memory as available physical memory increases.

Compounding address translation overhead is virtualization. Nested paging, the current technique for hardware supported memory virtualization, increases virtual memory overhead geometrically. Each address in a guest virtual address space must first be translated into a guest physical address. That guest physical address must then be translated into a host physical address. Since guest page tables on x86-64 store pointers in guest physical address form, a so-called *nested translation* can take as much as twenty-four memory accesses. Under nested paging, the overhead of virtual memory increases to up to 89% for real world workloads [1].

The primary focus of recent research to reduce this overhead has been to directly reduce the number of TLB misses by increasing the coverage of the TLB. Since TLBs are on the critical path of a memory access, their size is technology limited. Therefore, an increase in TLB coverage must come from an increase in the coverage of each particular entry, in other words, an increase in page size. Large pages have been shown to improve system performance for some workloads up to 30% as compared to using only small pages [5]. However, using large pages limits the ability of the operating system to control the allocation and protection of data. This can lead to wasted physical memory and I/O overheads. Additionally, fine-grained protection is required to emulate I/O devices in virtualization.

While this work has been successful at reducing the number of TLB misses, they can

never eliminate the problem entirely for two reasons. First, many environments like virtualization require fine grained allocation and protection, precluding the use of large pages. Additionally, while large pages increase TLB coverage, they only delay the problem. As physical memory sizes increase, the fraction of memory that can be covered by the TLB continues to shrink.

Comparatively less attention has been paid to reducing the cost of TLB misses once they happen. On x86-64, the high overhead of virtual memory is often traced to the large number of memory accesses required for a radix tree page table. In the common case, to translate a small page, it requires four accesses to the memory hierarchy per translation. However, since there is locality in virtual address use on a granularity larger than a page, there is reuse of these higher level page table entries. To exploit this, Intel and AMD have both introduced *MMU caches*, dedicated caches to serve upper levels of the page table. These devices can reduce the number of memory hierarchy accesses required to translate an address from four to nearly one.

AMD's MMU cache, the *page walk cache*, is essentially a very small L1 cache for page table entries. It holds 24 entries, uses an LRU replacement scheme and each entry is tagged by the page table entry's address in physical memory space. Intel's *paging structure caches* are a set of three caches, each responsible for caching a particular level of the table. This cache is tagged using portions of the virtual address translated by the page table entry. This allows the page walk hardware to "skip" cached levels of the tree. The differences in behavior between these very different designs has not been previously investigated.

Alternatively, other work has proposed replacing the radix table entirely. Some processors utilize a page table based on hashing, a decision supported by a significant body of research [6, 7]. In the ideal case, a hash table can reduce the number of required memory accesses to one. However, a hash table comes with significant overhead in terms of

tagging which can reduce the efficiency of page table entry caching. Literature comparing these designs is relatively old and has not investigated the impact of MMU caching on this tradeoff.

These systems all try to reduce the number of memory hierarchy accesses required per translation to one. However, this final memory access is the one most likely to miss in the data cache, so it can be extremely expensive. Nonetheless, no work has been done on reducing the impact of this final access. There is significant opportunity for performance increase if a system can be developed that predicts translations without any memory accesses at all. The use of a reservation based physical memory allocator can make such predictions possible. Such a system leaves significant alignment and contiguity in the mapping between virtual pages and physical pages. There is currently no system which exploits the predictability inherent in these underfilled reservations.

## 1.2 Contributions

This thesis begins by examining two current approaches to accelerating address translation: caching and alternative page table formats. We show that exploiting inter-page spatial locality is critical to performance. We performed the first comprehensive design space exploration and developed the first descriptive nomenclature of MMU caches. We divide five different designs into a two dimensional space. We show that caches tagged by virtual address parts (dubbed *translation caches*) outperform caches tagged by page table entry (PTE) physical address (*page table caches*). While their coverage is similar, translation caches are able to skip levels of the page table, reducing the number of memory hierarchy accesses required per walk.

We also analyze the tradeoffs between split and unified caches. This choice is a trade-off between isolating data of differing importance and adaptability to differing workloads.

Split caches preserve upper level entries at the expense of flexibility and die area. This hampers performance on large-memory applications. We present a novel replacement policy, Variable-Insertion Point LRU, that dynamically partitions the cache for a given workload. This replacement policy adapts to preserve entries that have high reuse. It allows a unified MMU cache of  $n + 1$  entries to perform as well as a split cache holding a total of  $3 \times n$  entries, even under pathological workloads.

Secondly, we compare the current x86-64 radix tree page table with competing formats. We examine an inverted page table, a clustered page table and a translation storage buffer. Our work is the first to compare a radix tree page table with a dedicated MMU cache against these alternate formats. We find that the overhead of tagging dramatically reduces the cachability of page table entries. Additionally, in a radix table, translations for pages adjacent in the virtual address space are adjacent in the page table. In a hash-based page table, however, adjacent translations are from random virtual addresses. This decreases the spatial access locality to the page table. The alternate page table formats require up to 4x as many DRAM accesses than the standard radix table. We conclude that the most important factors in page table design are the number of adjacent translations that can fit in a cache line and the total number of translations that can fit in the processor cache.

Finally, we present a novel device that exploits alignment and contiguity in the physical address space created by reservation based memory allocators. Our device can predict the physical address of a request that causes a TLB miss by using interpolation. This predicted address can be used for speculative execution while the page walk occurs in parallel, eliminating the latency of the TLB miss for these cases. In our simulations, it is able to remove a per-benchmark average of 56% of MMU-related DRAM accesses from the critical path of execution.

### **1.3 Organization**

This thesis is organized as follows. Chapter 2 introduces the x86-64 page table. It discusses the evolution of the table and its support for multiple page sizes. Chapter 3 discusses related work. Chapter 4 discusses and evaluates the different MMU cache designs. Each cache design is compared using a variety of different workloads. Chapter 5 evaluates competing page table formats using traces developed in Chapter 4. Chapter 6 introduces our novel speculative TLB design. Finally, we Conclude in chapter 7.



## CHAPTER 2

---

### x86 Address Translation

---

Much of the overhead of the x86 address translation system is due to its multilevel nature. The 48-bit virtual address space supported on current x86-64 processors is far too large to be supported by a simple flat page table. Therefore, it uses a four-level radix tree to efficiently store a sparsely utilized large address space. While this design is space efficient, it requires four sequential accesses to the page table per translation, causing a large latency penalty. This chapter examines the history and use of this page table design in detail while Chapter 4 examines hardware techniques to accelerate it.

#### 2.1 Page Walk Details

All x86 processors since the Intel 80386 have used a radix tree to record the mapping from virtual to physical addresses. Although the depth of this tree has increased, to accommodate larger physical and virtual address spaces, the procedure for translating virtual addresses to physical addresses using this tree is essentially unchanged. A virtual address is split into a page number and a page offset. The page number is further split into a sequence of indices. The first index is used to select an entry from the root of the tree, which may contain a pointer to a node at the next lower level of the tree. If the entry does contain a pointer, the next index is used to select an entry from this node, which may again contain a pointer to a node at the next lower level of the tree. These steps repeat until the selected entry is either

63:48	47:39	38:30	29:21	20:12	11:0
<i>se</i>	L4 idx	L3 idx	L2 idx	L1 idx	<i>page offset</i>

Figure 2.1 : Decomposition of the x86-64 virtual address.

invalid (in essence, a NULL pointer indicating there is no valid translation for that portion of the address space) or the entry instead points to a data page using its physical address. In the latter case, the page offset from the virtual address is added to the physical address of this data page to obtain the full physical address. In a simple memory management unit (MMU) design, this procedure requires one memory access per level in the tree.

Figure 2.1 shows the precise decomposition of a virtual address by x86-64 processors [8]. Standard x86-64 pages are 4KB, so there is a single 12-bit page offset. The remainder of the 48-bit virtual address is divided into four 9-bit indices, which are used to select entries from the four levels of the page table. The four levels of the x86-64 page table are named PML4 (Page Map Level 4), PDP (Page Directory Pointer), PD (Page Directory) and PT (Page Table). In this thesis, however, for clarity, we will refer to these levels as L4 (PML4), L3 (PDP), L2 (PD) and L1 (PT). Finally, the 48-bit virtual address is sign extended to 64 bits. As the virtual address space grows, additional index fields (*e.g.*, L5) may be added, reducing the size of the *se* field.

An entry in the page table is 8 bytes in size regardless of its level within the tree. Since a 9-bit index is used to select an entry at every level of the tree, the overall size of a node is always 4KB, the same as the page size. Hence, nodes are commonly called page table pages. The tree can be sparsely populated with nodes—if at any level, there are no valid virtual addresses with a particular 9-bit index, the sub-tree beneath that index is not instantiated. For example, if there are no valid virtual addresses with L4 index 0x03a, that entry in the top level of the page table will indicate so, and the 262,657 page table pages

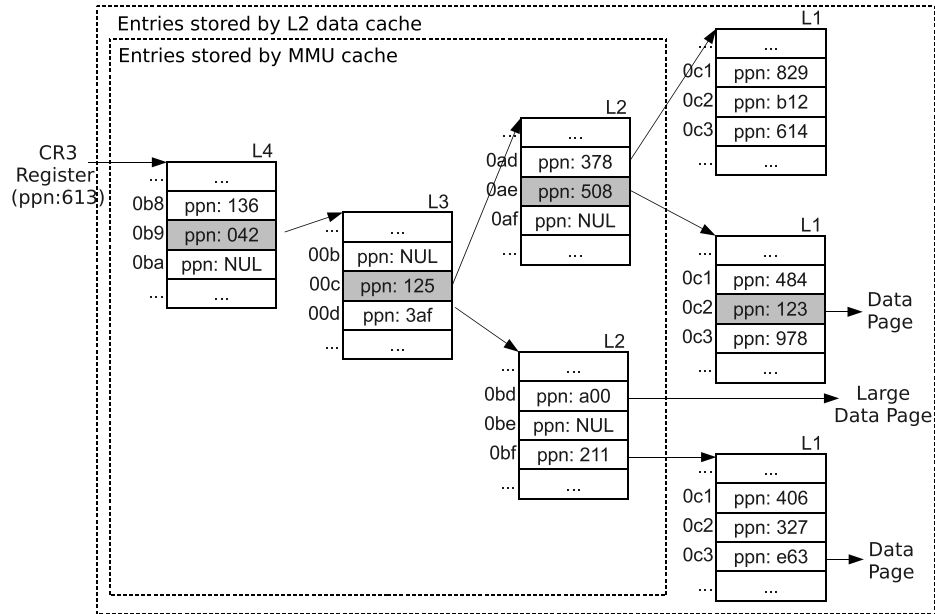


Figure 2.2 : An example page walk for virtual address (0b9 , 00c , 0ae , 0c2 , 016). Each page table entry stores the physical page number for either the next lower level page table page (for L4, L3, and L2) or the data page (for L1). Only 12 bits of the 40-bit physical page number are shown in these figures for simplicity.

(1 L3 page, 512 L2 pages, and 262,144 L1 pages) beneath that entry in the radix tree page table will not exist. This yields significant memory savings, as large portions of the 256 TB virtual address space are never allocated for typical applications.

Figure 2.2 illustrates the radix tree page table walk for the virtual address 0x0000 5c83 15cc 2016. For the remainder of the thesis, such 64-bit virtual addresses will be denoted as (*L4 index*, *L3 index*, *L2 index*, *L1 index*, *page offset*) for clarity. In this case, the virtual address being translated is (0b9 , 00c , 0ae , 0c2 , 016). Furthermore, for simplicity of the examples, only 3 hexadecimal digits (12 bits) will be used to indicate the physical page number, which is actually 40 bits in x86-64 processors.

As shown in the figure, the translation process for this address proceeds as follows.

First, the page walk hardware must locate the top-level page table page, which stores L4 entries. The physical address of this page is stored in the processor's CR3 register. In order to translate the address, the L4 index field (9 bits) is extracted from the virtual address and appended to the physical page number (40 bits) from the CR3 register. This yields a 49-bit physical address that is used to address the appropriate 8-byte L4 entry (offset 0b9 in the L4 page table page in the figure). The L4 entry may contain the physical page number of an L3 page table page (in this case 042). The process is repeated by extracting the L3 index field from the virtual address and appending it to this physical page number to address the appropriate L3 entry. This process repeats until the selected entry is invalid or specifies the physical page number of the actual data in memory, as shown in the figure. Each page table entry along this path is highlighted in grey in the figure. The page offset from the virtual address is then appended to this physical page number to yield the data's physical address. Note that since page table pages are always aligned on page boundaries, the low order bits of the physical address of the page table pages are not stored in the entries of the page table.

Given this structure, the current 48-bit x86-64 virtual address space requires four memory references to “walk” the page table from top to bottom to translate a virtual address (one for each level of the radix tree page table). As the address space continues to grow, more levels will be added to the page table, further increasing the cost of address translation. A full 64-bit virtual address space will require six levels, leading to six memory accesses per translation.

Alternatively, an L2 entry can directly point to a contiguous and aligned 2MB data page instead of pointing to an L1 page table page. In Figure 2.2, virtual address (0b9, 00d, 0bd, 123f5d7) is within a large page. This large-page support greatly increases maximum TLB coverage. In addition, it lowers the number of memory accesses to locate one of these pages from four to three. Finally, it greatly reduces the number of total page

table entries required since each entry maps a much larger region of memory.

## CHAPTER 3

---

### Related Work

---

Work on reducing the performance penalty of virtual memory can be divided into techniques that reduce the impact of TLB misses and those that reduce their frequency. Previous work on reducing the impact of TLB misses have primarily investigated alternative page table formats and caching, as examined in this thesis. Systems that reduce TLB miss frequency have been primarily related to large page use and TLB prefetching. Our work is generally complementary to these techniques.

#### 3.1 Caching

Some early work on caching page table entries was done before the introduction of AMD and Intel's MMU caches. This work was targeted at accelerating software TLB miss handling. Bala *et al.* introduced a software cache for page table entries [9]. This cache is read by the software page fault handler and manages entries in physical memory to avoid cascading TLB misses that come from reading page table entries in virtual memory space. Wu and Zwanepoel expanded this to a hardware/software design [10]. They propose a dedicated cache to handle L2 page table entries. If a translation hits in their structure, the MMU loads the L1 entry directly, as in the caches presented in this thesis. If the translation misses, a software fault is triggered. We extend this proposal into a device that caches all upper levels in Chapter 4.

McCurdy *et al.* investigated the TLB performance of a wide variety of high-performance computing applications over different page sizes [2]. They show that this class of application can have significant overhead from memory management. Changing page sizes can change application performance by up to 50% in many cases. They also show that the HPCC and SPEC benchmarks do not necessarily represent these applications, and are poor analogues for choosing page sizes with.

The authors point out the importance of the L2 data cache in storing page table entries. Applications were profiled with performance counters on actual hardware. This reveals that applications that use large pages can show improved performance even in the face of *decreased* TLB hit rates. This is due to the shallower, and therefore smaller, page table when using large pages.

Bhargava *et al.* first described the AMD page walk cache and their extensions to it to support virtualization. This cache is described in detail in Chapter 4.

## 3.2 Alternate page table formats

Jacob and Mudge [11] compare five production and proposed systems with different memory management systems. They conclude that the x86-32 MMU has the highest performance because it works entirely in hardware, without any software handling. The precise interrupts required by software MMUs, such as those on MIPS, present an overhead that is not hidden by caching. We examine these interrupts in the context of SPARC's Translation Storage Buffer in Chapter 5. They also show that MMU related memory accesses can cause higher than expected cost due to user program and data being evicted by page table entries. This effect further emphasizes the importance of efficient storage of page table entries.

In terms of space, a radix tree-based page table can be an inefficient representation for a large, sparsely-populated virtual address space. Liedtke introduced *Guarded Page Tables* to

address this problem [12]. In particular, Guarded Page Tables allow for path compression. If there is only one valid path through multiple levels of the tree, then the entry prior to this path can be configured such that the page walk will skip these levels.

Talluri and Hill presented alternate forms of inverted page tables called clustered and subblocked page tables [6, 7]. These designs associate a power of two number of adjacent translations with a single virtual tag. The high bits of a virtual address are compared against the tag while the low bits select a particular subblock. We simulate a similar system in Section 5.1 and compare it to both traditional inverted page tables and cached radix tree tables. They also present support for a subblock TLB which uses the same tag stored in the page table. Each entry in their TLB stores multiple translations with a common virtual tag. This provides some of the benefits of large pages without requiring operating system modifications beyond support for the new page table format.

This technique reduces the space overhead of the page table by reducing the number of tags that are required per translation. However, it also increases the number of translations in the page table that can fit in a processor data cache (or a single line of the cache), which we show in Chapter 5. We conclude that such a system is not as efficient as the radix tree page table.

### **3.3 Reducing TLB miss frequency**

Talluri and Hill ([6]) first proposed a reservation based system for using clustered page tables. Navarro *et al.* ([5]) extended this idea into a practical memory allocator for superpages and implemented it under FreeBSD. Most critically, this extension developed the reclamation of underfilled superpage reservations. While these works are trying to create full reservations, as a by-product, they also create contiguity in the page table that is exploited in this thesis. Additionally, their work extended the system to support multiple page



sizes.

Early work on implementing superpage support in general-purpose operating systems used a static page size determination. IRIX ([13]), Solaris ([14]) and HP-UX ([15]) allocate superpages at page fault time. An process tells the operating system that a region of its address space should be allocated using large pages. These operating systems then allocate and map the entire page at once, unlike the reservation and promotion process used by FreeBSD. Since these systems do not place small pages in a reservation, they will not benefit from the techniques described in this work.

Romer *et al.*, ([16]) propose the creation of superpages by moving existing pages, previously scattered throughout physical memory, into contiguous blocks. While this process may be prohibitively expensive for very large superpages, it may have more success with the architecture described here that does not require a full reservation.

Saulsbury *et al.* propose a prefetching scheme for TLBs that preload pages based recently accessed pages [17]. Unlike the techniques presented in this thesis, their techniques require page table modification. More recent work [18, 19] has proposed architecturally independent prefetching techniques based on access patterns and inter-core cooperation.

These techniques all focus on reducing the frequency of TLB misses while our work focuses on reducing the cost of servicing a TLB miss. Both types of techniques are complementary and could easily be combined.

## CHAPTER 4

---

### MMU Caching

---

#### 4.1 Introduction

This chapter explores the design space of memory-management unit (MMU) caches for accelerating virtual-to-physical address translation in processor architectures, like x86-64, that implement paged virtual memory using a radix tree for their page table. In particular, these caches accelerate the page table walk that occurs after a miss in the Translation Lookaside Buffer (TLB). In fact, a hit in some of these caches enables the processor to *skip* over one or more levels of the tree and, in the best case, access only the tree's lowest level.

For several generations of x86 processors, from the Intel 80386 to the Pentium, the page table had at most two levels. Consequently, whenever a TLB miss occurred, at most two memory accesses were needed to complete the translation. However, as the physical and virtual address spaces supported by x86 processors have grown in size, the maximum depth of the tree has increased, first to three levels in the Pentium Pro to accommodate a 36-bit physical address within a page table entry, and more recently to four levels in the AMD Opteron to support a 48-bit virtual address space. In fact, with each passing decade since the introduction of the 80386, the depth of the tree has grown by one level.

Recent work has shown the impact of TLB misses on overall system performance ranges from 5-14% for nominally sized applications, even in a non-virtualized environ-

ment [1]. As the application’s memory footprint increases, TLB misses have a significantly larger impact on performance, approaching 50% in some cases [2]. Although the use of large pages can lessen this impact, with further increases in the memory footprint their effectiveness declines. Therefore, both AMD and Intel have implemented MMU caches for page table entries from the higher levels of the tree [20, 1]. However, their caches have quite different structure. AMD’s Page Walk Cache stores page table entries from any level of the tree, whereas Intel implements distinct caches for each level of the tree. Also, AMD’s Page Walk Cache is indexed by the physical address of the cached page table entry, whereas Intel’s Paging-Structure Caches are indexed by portions of the virtual address being translated. Thus, in this respect, the Page Walk Cache resembles the processor’s data cache, whereas the Paging-Structure Caches resemble its TLB.

This chapter’s primary contribution is that it provides the first comprehensive exploration of the design space occupied by these caches. In total, it discusses five distinct points in this space, including three new designs. Specifically, it presents the first head-to-head comparison of the effectiveness of these designs. In general, the results of this comparison show that the *translation caches*, which store partial translations and allow the page walk hardware to skip one or more levels of the page table, are the best. In addition, the new translation caches that are introduced by this chapter are better than the existing caches in many situations and workloads.

This chapter is organized as follows. Section 4.2 describes the design space, identifying the essential differences between AMD’s Page Walk Cache, Intel’s Paging-Structure caches, and the new structures developed in this chapter. Section 4.3 qualitatively compares these structures, and Section 4.5 describes this chapter’s methodology for quantitatively comparing them. Section 4.6 presents quantitative simulation results of their effectiveness as compared to one another. Finally, Section 4.8 summarizes this chapter’s conclusions.

## 4.2 Caching page walks

While radix-tree page tables require many accesses to translate a single address, the accesses to the upper level page table entries have significant temporal locality. Walks for two consecutive pages in the virtual address space will usually use the same three upper level entries, since the indices selecting these entries come from high-order bits of the virtual address, which change less frequently.

While the MMU does access the page table through the memory hierarchy, it only has access to the L2 data cache in at least one major commercial x86 design [1]. Since the L2 data cache is relatively slow on modern CPUs, accessing three upper-level page table entries on every page walk will incur a penalty of several tens of cycles per TLB miss, even if all entries are present in the L2 data cache.

Therefore, the x86 processor vendors have developed private, low-latency caches for the MMU that store upper level page table entries [1, 20]. In this section, we describe the design space and provide a nomenclature for the different tagging and partitioning schemes used by these MMU caches.

MMU caches may store elements from the page table tagged by their physical address in memory, as a conventional data cache might. We call such MMU caches *page table caches*. Examples include AMD’s *Page Walk Cache* and the L2 data cache, although it is not private to the MMU. Alternatively, MMU caches can be indexed by parts of the virtual address, like a TLB. We call such MMU caches *translation caches*. Intel’s *Paging-Structure Caches* are translation caches.

For either of these tagging schemes, elements from different levels of the page table can be mixed in a single cache (a *unified* cache), or placed into separate caches (a *split* cache). Finally, each cache entry can store an entry from one level along the page walk, or it can

store an entire path (a *path* cache).

#### 4.2.1 Page table caches

The simplest example of a page table cache is the processor's L2 data cache. The page walker generates a physical address based upon the page table page to be accessed and an index from the virtual address. This physical address is then fetched from the processor's memory hierarchy starting with the L2 data cache.

Page table caches use this same indexing scheme. Elements are tagged with their physical address in the page table. These tags are the size of the physical page number plus the size of one page table index. L1 entries are not cached in any of the designs presented here (since the TLB itself caches those entries).

Base Location	Index	Next Page
125	0ae	508
042	00c	125
613	0b9	042
...	...	...

Figure 4.1 : An example of the contents of a UPTC. Each entry is tagged with the address of a page table entry, consisting of the 40-bit physical page number of the page table page and a 9-bit index into it. The entry then provides a 40-bit physical page number for the next lower level page table page. (Only 12 bits of the physical page numbers are shown, for simplicity.)

#### Unified Page Table Cache (UPTC)

The simplest design for a dedicated page table cache is a single, high-speed, read-only cache for page table entries, tagged by their physical address in memory. Entries from different levels of the page table are mixed in the same cache, all indexed by their physical address. Such a cache is analogous to a private, read-only L1 data cache for page table

	Base Location	Index	Next Page
L2 entries	125 ...	0ae ...	508 ...
L3 entries	042 ...	00c ...	125 ...
L4 entries	613 ...	0b9 ...	042 ...

Figure 4.2 : An example of the contents of a SPTC. Each entry holds the same tag and data as in the UPTC.

entries. However, like a TLB, coherence between this cache and the page table can be maintained by software with little overhead. AMD's Page Walk Cache has this design [1].

Figure 4.1 shows an example of the Unified Page Table Cache (UPTC) after the MMU walks the page table to translate the virtual address (0b9, 00c, 0ae, 0c2, 016). If the MMU subsequently tries to translate the virtual address (0b9, 00c, 0ae, 0c3, 103), the page walk will begin by looking for the page table entry 0b9 in the L4 page table page (located at 613 and referenced by the CR3 register). Since this page table entry is present in the UPTC, it does not need to be loaded from the memory hierarchy.

This entry indicates that the L3 page table page has physical page number 042. The same process is then repeated to locate the L2 and L1 page table pages. Once the address of the L1 page table page is found, the appropriate entry is loaded from memory to determine the physical page address of the desired data.

Without a page table cache, all four of these accesses to page table entries would have required a memory access, each of which may or may not hit in the L2 data cache. In contrast, with the page table cache, the three top entries hit in the private page table cache, and only one entry (the L1 entry) requires a memory access, which may or may not hit in

the L2 data cache.

### **Split Page Table Cache (SPTC)**

An alternate design for the page table cache separates the page table entries from different levels into separate caches. Figure 4.2 illustrates a Split Page Table Cache (SPTC). In this design, each individual entry contains the same tag and data as it would in the unified page table cache. The primary difference is that each page table level gets a private cache, and entries from different levels do not compete for common slots.

#### **4.2.2 Translation caches**

As an alternative to tagging cache entries by their physical address, they can be tagged by their indices in the virtual address. An L4 entry will be tagged by the 9 bit L4 index, an L3 entry with the L4 and L3 indices, and an L2 entry with the L4, L3, and L2 indices. We call this device a *translation cache*, because it is storing a partial translation of a virtual address.

With this tagging scheme, data from one entry is not needed to lookup the entry at the next lower level of the page table. All of the lookups can be performed independently of each other. In the end, the MMU will select the entry that matches the longest prefix of the virtual address because it allows the page walk to skip the most levels.

### **Split Translation Cache (STC)**

Like an SPTC, the Split Translation Cache (STC) stores entries from different levels of the page table in separate caches. However, as shown in Figure 4.3, the STC uses a different way of tagging the entries. The Intel *Paging-Structure Caches* [20] exemplify the STC organization.

	L4 index	L3 index	L2 index	Next Page
L2 entries	0b9 ...	00c ...	0ae ...	508 ...
L3 entries	0b9 ...	00c ...		125 ...
L4 entries	0b9 ...			042 ...

Figure 4.3 : An example of the contents of a STC. Each index is 9-bits, and the data holds a 40-bit physical page number of the next page table level. An entry in the L2 cache must match on all three indices, an entry in the L3 must match on two and the L4 on one.

L4 index	L3 index	L2 index	Next Page
0b9	00c	0ae	508
0b9	00c	xx	125
0b9	xx	xx	042
...	...	...	...

Figure 4.4 : An example of the contents of a UTC. An “xx” means “don’t care”.



The example in Figure 4.3 shows the split translation cache after the MMU walks the page table to translate the virtual address (0b9, 00c, 0ae, 0c2, 016). If the MMU subsequently starts to translate the virtual address (0b9, 00c, 0dd, 0c3, 929), it will attempt to locate the L1, L2 and L3 page table pages in their corresponding caches using portions of the virtual address. The location of the L3 page table page would be stored in the L4 entry cache and tagged by the L4 index, (0b9). Similarly, the location of the L2 page table page would be stored in the L3 entry cache and tagged by the L4 and L3 indices, (0b9, 00c). Finally, the location of the L1 page table page would be stored in the L2 entry cache and tagged by the L4, L3 and L2 indices, (0b9, 00c, 0dd).

These searches can be performed in any order, and even in parallel. In the above example, the cache can provide the location of the appropriate L3 and L2 page table pages, but not the L1 page table page, as (0b9, 00c, 0dd) is not present in the L2 entry cache. Ultimately, the MMU would use the (0b9, 00c) entry from the L3 entry cache because it allows the page walk to begin further down the tree, at the L2 page table page.

### **Unified Translation Cache (UTC)**

Just as the page table caches can be built with either a split or a unified organization, a Unified Translation Cache (UTC) can also be built. Moreover, just like the UPTC, the UTC mixes elements from all levels of the page table in the same cache.

Figure 4.4 shows the UTC after the MMU walks the page table to translate the virtual address (0b9, 00c, 0ae, 0c2, 016). If the MMU subsequently starts to translate the virtual address (0b9, 00c, 0dd, 0c3, 929), it will first look in the UTC for the physical page numbers of the L1, L2 and L3 page table pages. As with the previous example that used the STC, the MMU finds two matching entries in the UTC. Ultimately, the MMU decides to use the UTC's second entry, which is an L3 entry that has the L4 and

L3 indices (0b9 , 00c) as its tag, because this tag matches the longest prefix of the virtual address. Thus, the MMU can skip the L4 and L3 page table pages and start walking from the L2 page table page.

### 4.2.3 Translation-Path Cache (TPC)

Note that in the UTC example in Figure 4.4, the tags for the three entries representing a single path down the page table all have the same content. The L4 and L3 entries use less of the virtual address than the L2 entry does, but the fragments that they do use are the same. Consequently, it is possible to store all three physical page numbers from this example in a single entry. In such a *Translation-Path Cache* (TPC), a single entry represents an entire path, including all of the intermediate entries, for a given walk instead of a single entry along that walk.

L4 index	L3 index	L2 index	L3	L2	L1
0b9	00c	0ae	042	125	508
...	...	...	...	...	...

Figure 4.5 : An example of the contents of the TPC after the virtual address (0b9 , 00c , 0ae , 0c2 , 016) is walked. The TPC holds three 9 bit indices, as the translation caches do, but all three 40-bit physical page numbers are stored for all three page table levels.

The example in Figure 4.5 shows the TPC after the MMU walks the page table to translate the virtual address (0b9 , 00c , 0ae , 0c2 , 016). All data from that walk is stored in one entry. If the MMU subsequently starts to translate the virtual address (0b9 , 00c , 0ae , 0c3 , 929), the entry referencing the L1 page table page is discovered just as it would have been in the unified translation cache. Specifically, the MMU finds the entry in the cache with the tag (0b9 , 00c , 0ae) and reads the physical page number 508 of the L1 page table page from this entry.

If the MMU later starts to translate the virtual address (0b9, 00c, 0de, 0fe, 829), this address shares a partial path (0b9, 00c) with the previously inserted entry. Therefore, the translation-path cache will provide the physical address of the appropriate L2 page table page.

#### 4.2.4 Design space summary

In summary, the caches described in this section fit into the following two-dimensional design space (annotated with the section number in which each design is described):

	Unified	Split	Path
Page Table Cache	4.2.1	4.2.1	N/A
Translation Cache	4.2.2	4.2.2	4.2.3

The unified page table cache (UPTC) is the design that appears in modern AMD x86-64 processors. The split translation cache (STC) is the design that appears in modern Intel x86-64 processors. The remaining three designs have been introduced in this chapter.

Note that there is no useful page table counterpart to the translation-path cache. This is a direct result of the indexing scheme. While a “path” of physical addresses could be stored as an MMU cache index, it would have to be searched sequentially because the MMU cannot create a path of physical page numbers directly from the virtual address. It must look up each physical page in turn. Therefore, storing complete paths would yield no benefit over the other page table cache designs.

### 4.3 Design comparison

All of the designs presented in the previous section are able to accelerate page walks by caching information from the upper levels of the page table. However, these designs have

differences in their indexing, partitioning, coverage, and complexity. This section discusses the effects of these differences.

### **4.3.1 Indexing**

The indexing scheme determines how the cache is searched. Cache indices can be derived from the physical addresses of components of the page table or they can be derived from the virtual address and correspond to the levels of the page table.

Page table caches use the physical addresses of the page table entries as indices. In fact, they operate identically to any physically indexed cache—a UPTC is essentially another data cache in the memory hierarchy dedicated to the page table. The MMU will generate a physical address for the page table entry at each stage of the page walk, and that address will be used as an index into the appropriate page table cache. While this leads to a simple design, it requires the cache lookups to occur in a top-down order. The result of the L4 entry search is required before the L3 entry search can begin, because the L4 entry gives the physical page number of the L3 page table page, which is needed to generate the physical address of the L3 page table entry. Similarly, the L2 search is dependent on the result of the L3 search. In the case where the cache holds all three entries (L4, L3, and L2), the cache must be accessed three times to generate the physical address of the L1 page table entry.

In contrast, translation caches use components of the virtual address as indices. For example, the TLB is an L1 translation cache that uses the virtual page number as its index. In general, for translation caches, the MMU uses a prefix of the virtual page number as the index. This allows the translation caches to be searched in any order (L4 first, L2 first, or in parallel). Thus, on a TLB miss, the L2 translation cache can immediately be searched. Upon a hit which yields the L2 translation, the address of the L1 page table entry can be computed immediately. If no L2 translation is available, the L3, then L4, translation caches

can be searched. Upon a hit, the page walk would begin at that point in the tree.

#### **4.3.2 Partitioning**

MMU caches can either be unified or partitioned, or they can store complete path information. The partitioning of the cache determines how the entries of the cache are allocated to different levels of the page table. This effectively determines how well the entries from different levels are isolated from each other.

The impact of the partitioning scheme largely depends on whether the application densely or sparsely utilizes its virtual address space. For applications that densely use their virtual memory, a few L4 and L3 entries are heavily utilized and there is significant reuse of the L2 entries. In contrast, for applications that sparsely use their virtual memory, there will be little reuse for L2 entries, but a larger number of L4 and L3 entries will exhibit reuse. The partitioning strategy and replacement policy determine how these entries will compete for slots, which can have a significant impact on the effectiveness of the MMU cache.

For applications that densely use their virtual memory, there will be many more L2 entries than upper level entries in use. However, for a page table cache design, these upper level entries are critical for translation performance. If a random replacement scheme is used in conjunction with a unified cache, these important entries can be frequently replaced, resulting in memory accesses to entries at or near the top of the table. However, if entries from different levels are kept in separate caches, a random replacement policy is less detrimental.

For applications that sparsely use many gigabytes of virtual memory, L2 entries will have very little reuse, and effective caching of L3 entries is critical. In a split entry cache, a static allocation of entries to each level must be made. If this allocation is optimized for

small applications, it will have many more L2 entries than L3 entries, harming performance in this situation. Moreover, if an application makes heavy use of large pages and limited use of small pages, the dedicated L2 entries will be of little or no use. In contrast, in a unified cache, the allocation happens dynamically, but recently accessed L2 entries that will not be reused might evict L3 entries that might otherwise be reused. A level-aware replacement policy can help to avoid this.

The Greedy Dual algorithm is a popular content-aware replacement scheme [21]. This algorithm will replace recently used entries early if they are easier to reload into the cache. This scheme can be adapted for MMU caches by preferentially replacing lower-level entries with upper-level entries, thus reducing conflict between entries of high and low reuse. While Greedy Dual is not an algorithm that can easily be implemented in hardware, it is possible to implement a similar algorithm with minimal modification to an LRU cache.

In our modified LRU algorithm, entries from lower levels of the page table are inserted into the LRU queue at a recency position behind the most-recently-used position. If these lower level entries are reused, they are promoted to the most recently used position. However, if they are not reused, the portion of the cache in which lower level entries compete with upper level entries is small. These positions can be fixed, for simplicity, or they can change to adapt to different workloads. We propose a *variable insertion-point LRU replacement policy* whereby entries from lower levels of the cache are inserted into a recency position below the most recently used position that is proportional to the current number of upper level entries stored. For example, if there are two L4 entries and six L3 entries currently in the cache, a new L2 entry is inserted in the ninth most recently used position in the cache.

The path translation cache avoids these partitioning problems, as each slot holds an entry from all levels. This prevents the competition for slots while not requiring a static

allocation of slots to levels. However, since this cache may hold many paths with the same upper level entries, its effective capacity for holding upper level entries is less than three equally sized split entry caches.

### 4.3.3 Coverage

Characterizing the coverage of an MMU cache is not straightforward. In particular, the exact meaning of coverage for an MMU cache must first be considered. For example, suppose an address translation hits on an L3 entry in a page table or translation cache but does not hit on an L2 entry. In this case, the translation was accelerated by the MMU cache, but nonetheless required a memory access to fetch the page table's L2 entry. Thus, it is arguable whether or not the MMU cache provided coverage. We take the strict position that coverage means that no memory accesses were required to fetch L4, L3, or L2 page table entries.

In general, with the same number of entries, translation caches are able to cover a larger portion of the address space than page table caches. The reason is that a translation cache can make more efficient use of its entries than a page table cache. For a page table cache to provide coverage it must simultaneously hold an L4, L3, and L2 entry, whereas a translation cache can provide coverage with only an L2 entry. In other words, the translation cache may be able store additional L2 entries in place of the L4 and L3 entries that are required to provide coverage in the page table cache.

It can be shown that regardless of access pattern, a TPC storing  $n$  paths can provide any entry stored in a  $n$  entry unified cache. Additionally, a split page table cache of  $3 \times n$  entries can provide any entry stored in an  $n$  path TPC. These proofs are given in Appendix A.

When the application is simply too large for the MMU cache to provide full coverage,

		Unified	Split	Path
Page Table	C	1	$l - 1$	
	T	$p - 3$	$\{p - 3, \dots, p - 3\}$	N/A
	D	$p - 12$	$\{p - 12, \dots, p - 12\}$	
Translation	C	1	$l - 1$	1
	T	$(l - 1) \cdot n$	$\{n, \dots, (l - 1) \cdot n\}$	$(l - 1) \cdot n$
	D	$p - 12$	$\{p - 12, \dots, p - 12\}$	$3(p - 12)$

Table 4.1 : The number of caches (C), the number of tag bits per entry (T), and the number of data bits per entry (D) for each design.

the unified caches with our proposed VI-LRU replacement policies, the split caches, and the TPC are able to accelerate translations for more of the address space than the unified caches with conventional replacement policies. This is due to the fact that upper level entries are maintained longer in these caches. Since these effects are highly dependent on workload, the relative hit rates of the cache designs are studied experimentally in Section 4.6.

## 4.4 Implementation

In this section, we examine the relative costs of implementing the caches discussed in Section 4.2. While the design of caches in general is a well-studied field, each design has differing requirements. We begin by comparing these requirements on a high level, then we conclude by examining some specific circuit details unique to MMU caches.

All of the organizations examined are effectively fully associative caches, which can be implemented by a CAM array to match the tags and a RAM array to store the data entries. However, the different cache organizations have different tag and data widths, and will potentially require differing numbers of entries to achieve similar hit rates. These factors will lead to different implementation complexities for the different organizations.

Table 4.1 shows the number of caches, tag bits per entry, and data bits per entry that



		Unified	Split	Path
Page Table	C	1	3	
	T	49	{49, 49, 49}	N/A
	D	40	{40, 40, 40}	
Translation	C	1	3	1
	T	27	{9, 18, 27}	27
	D	40	{40, 40, 40}	120

Table 4.2 : MMU cache parameters for x86-64 processors. We have used the architectural definition of physical address width, 52 bits [8]. Actual implementations may use fewer bits.

are needed for each organization. These characteristics are parameterized by the number of levels of address translation,  $l$ , the number of bits in a physical address,  $p$ , and the number of offset bits in a page table index for a particular level,  $n$ . In x86-64 processors,  $l = 4$ ,  $p = 52$ , and  $n = 9$ , which leads to the values shown in Table 4.2.

It should be noted that for current architectural parameters, translation caches require significantly smaller tags. This will make a translation cache smaller and more power efficient than an equivalent page table cache, as the CAM array is likely to dominate the power and area required by the structure.

#### 4.4.1 CAM Bypass

Fully-associative caches are not a new invention. However, the TPC and UTC add some additional complexity to the design because portions of the virtual address must be selectively ignored during a search. In this section we present some conjectural modifications to a standard fully-associative cache to support this constraint. We use the cache model from the CACTI tool from HP Labs [22] as our baseline in Figure 4.6.

This design uses dynamic logic to compare the input address with each line in parallel. A closeup of the individual bit-comparison logic is shown in Figure 4.7. The match line

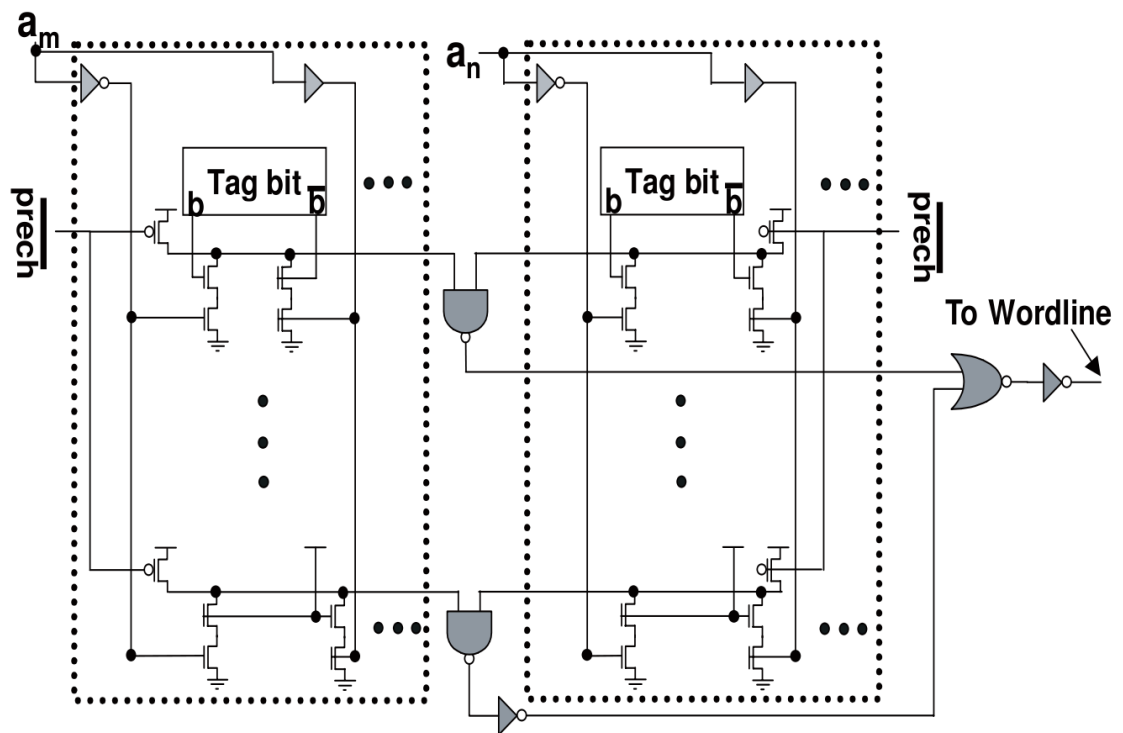


Figure 4.6 : An example fully associative cache design from [22].

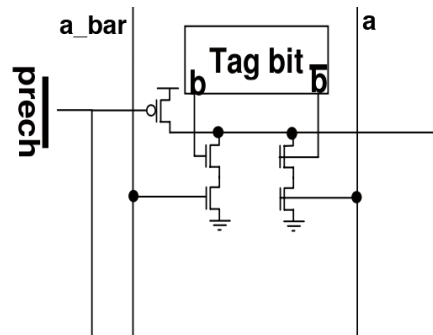


Figure 4.7 : A closeup of the dynamic bit match logic.

is initially precharged high. If the tag bit is a one, the right side pull-down network is activated by the upper NMOS transistor. If the tag bit is a zero, the left side pull-down network is activated.

If the address bit (the bit being searched for) is a one, the bottom left NMOS transistor is activated, allowing the match line to drain if the tag bit is a zero. Similarly, if the address bit is a zero, the bottom right NMOS transistor is activated, allowing the match line to drain if the tag bit is a one. Otherwise, the address bit matches the tag bit and the match line remains high.

This comparison can be easily bypassed with the introduction of a third transistor to the network shown in Figure 4.8. A bypass line masks a bit off from being included in the search and thus ensures that bit will always match. When *bypass* is high, the new NMOS transistor prevents the pull-down network from draining the match line. Therefore, it will remain charged high. This modification will increase area somewhat and will increase delay because the resistance of the pull down network is increased.

The TPC can use this facility to mask off lower level indices when searching for upper level page tables. Initially, all bypass lines are low. If that search fails, the TPC uses the L4 and L3 index to search for an L3 entry. Here, all the bits corresponding to the L2 index

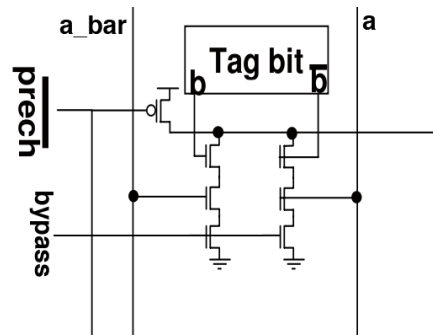


Figure 4.8 : The dynamic bit match logic with bypass line.

are bypassed. Similarly, L2 and L3 indices are masked off when searching for L4 entries. The UTC can use these bits, in conjunction with additional tag bits, to always bypass lower order bits for lines containing upper level entries. This allows the UTC to act somewhat like a TCAM, in that certain bits are ignored during a search. Unlike a TCAM, however, the ignored bits can not be arbitrarily specified, they must correspond to either masking off the L2 offset or the L2 and L3 offsets. This constraint reduces the number of additional tag bits that would be required for a full TCAM.

#### 4.4.2 VI-LRU shift register implementation

AMD has described their page walk cache as having a “least recently used replacement policy” [1]. While their exact implementation is not described in their paper, one possible implementation of an LRU cache is to use a shift register. All the tag bits in the cache structure (see Figure 4.6) are connected such that one loads into the next, as shown in Figure 4.9. Addresses are inserted into one end of the chain and old addresses propagate down. This requires each cache entry of  $n$  bits to be connected to the next entry with  $n$  wires.

Our VI-LRU replacement scheme requires data to be inserted at arbitrary points inside

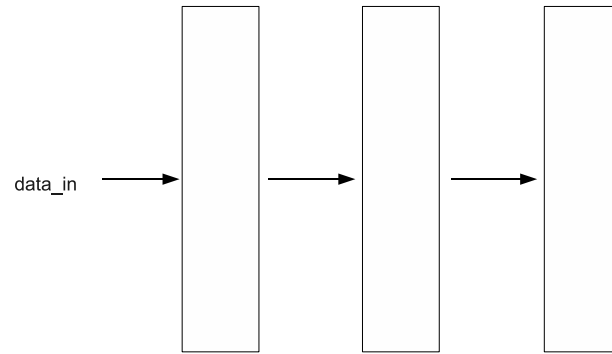


Figure 4.9 : An example implementation of an LRU replacement scheme.

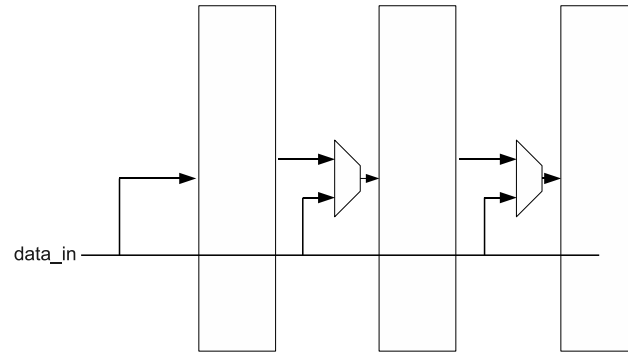


Figure 4.10 : An example implementation of the VI-LRU replacement scheme.

the chain. One possible implementation of this is shown in Figure 4.10. In this design, the new value is routed not only to the first entry, but to every entry in the cache. This doubles the number of wires required for the chain, one for the old value and one for the new one. At the input to each register, a multiplexer selects which cache entry the new value is to be inserted into. This multiplexer must be activated by global replacement logic.

For an  $n$  entry,  $m$  bit cache, this modification adds  $m(n - 1)$  multiplexers,  $m(n - 1)$  wires to route new data throughout the cache and  $m(n - 1)$  wires to select the multiplexors. This increases the area of the cache and the delay required to update an entry, but it does not add any circuitry to the match logic.

## 4.5 Methodology

The MMU cache architectures presented in Section 4.2 were evaluated by running application memory traces through a memory system simulator. The trace-based approach is warranted here for two reasons. First and foremost, the number of memory references required for a page walk is effectively independent of all architectural parameters except for the MMU and the L2 cache organization. A cycle-accurate simulation would have presented a more limited view of the differences for a single point in the processor design space. Second, from a practical standpoint, it would have been nearly impossible to run the types of large memory footprint applications that benefit most from these structures on a slow cycle-accurate simulator.

### 4.5.1 Application Memory Traces

The AMD SimNow [23] platform simulator was used to run various benchmarks under FreeBSD 8.0-Release for x86-64. A custom analyzer plugin to SimNow records each virtual memory access made by the simulated system. This trace includes all memory loads and stores made by the tested operating system and processes, but it does not include instruction or page table loads. TLB and MMU invalidations are included in the trace by monitoring the value of the CR3 register, which must change on a context switch. Finally, the plugin counts the total number of instructions executed during the trace.

Virtual memory access traces were captured from various applications, including the SPEC CPU2006 floating-point suite [24], SPECjbb2005 [25], ASCI Sweep3d [26] and HPCC RandomAccess [27]. However, not all of the benchmarks in the SPEC CFP2006 suite could be compiled with the standard tool chain in FreeBSD 8.0, so *soplex*, *calculix* and *wrf* are not included in this study. SPECjbb2005 was run on one warehouse, and

Sweep3d was run on a 150x150x150 grid.

#### 4.5.2 Memory System Simulation

A custom memory system simulator was built to simulate the various MMU cache designs. The simulator includes an MMU that closely resembles the L1 and L2 TLBs in the AMD Opteron [1]. It consists of a 64-entry, fully-associative L1 TLB with random replacement, and a 512-entry, 4-way set associative L2 TLB with LRU replacement. Furthermore, the simulator is able to model all five cache designs described in Section 4.2. The simulator stores tags (virtual addresses), but not data (physical addresses), to eliminate any operating system dependent behavior from the simulation. This simplifies the design of the simulator and generalizes the results. Unless otherwise specified, the simulator divides all memory into 4KB pages.

A 1MB L2 cache was included in the model, simulated using the Dinero IV cache simulator [28]. Both application data accesses and MMU page table accesses are simulated using a shared L2 cache model. In general, instruction loads are not included in this study. The cache parameters were based on the same AMD Opteron processor that was the basis for the TLB parameters. The L1 cache was not simulated, since the page walk hardware does not use it on the Opteron.

While our simulation environment did not permit us to directly measure power and system performance, the reduction in memory accesses we directly measure here should translate directly into reduced interconnect power consumption and latency. Recall that previous work has shown the uncached system performance impact of unvirtualized TLB misses to be up to 14% for nominally sized applications [1] and up to 50% for large applications [2].

### 4.5.3 Synthetic Application Memory Traces

To study the behavior of an application that uses more virtual memory than we can practically trace on our real machines, a trace synthesizer was developed that simulates the memory access pattern of an in-memory database, performing a hash join. Such joins are common, and the performance of large joins is representative of overall database performance [3].

The simulated join is an inner join on two equally sized tables, A and B. The hash join process starts by creating a hash table containing the entries of B, using an open addressing collision resolution scheme. The database then iterates through A, checking to see if each entry is present in the hash table. The result is then placed in an output table [29].

Since the simulation is designed to scale to arbitrary sizes, the simulation works probabilistically rather than operating on a real data set. First, an element is read from the region of memory holding table A. Then, a random element is read from the region of memory storing the hash table, since the hash function will uniformly distribute accesses throughout the table. After the first element is read, a second element is sometimes also read, based on the probability of a hash collision. The collision probability was derived from the expected collision chain length [30]. Finally, an element is written to the result table, and the process repeats itself with the next consecutive element of table A.

## 4.6 Cache design simulations

This section evaluates the five different MMU cache organizations using a wide variety of applications. The TLB miss penalty, structure sizing, and replacement policies are explored. The results show that the unified translation cache with a modified LRU replacement scheme is the best design for the entire range of applications. For the small bench-



marks, this cache design is able to reduce the number of memory accesses required per TLB miss from 4 without a dedicated cache to 1.13. It also adapts dynamically to large applications, avoiding the conflicts present in traditional unified caches without the static partitioning required in split caches.

#### **4.6.1 TLB miss penalty**

The purpose of any MMU cache is to lessen the penalty of a TLB miss and the cost of walking the page table. This penalty can be broken down into the number of accesses to the private MMU cache and the number of memory hierarchy accesses required per TLB miss. Without a private cache, there will be four memory hierarchy accesses per walk, one per level. These memory accesses can be further broken down into L2 data cache hits and DRAM accesses, which are far more costly.

#### **Small Memory Applications**

Even applications that use a modest amount of memory can have frequent TLB misses. Table 4.3 shows the frequency of TLB misses for each of the SPEC CFP2006 benchmarks, the SPEC JBB2005 Java server benchmark and the ASCI Sweep3d benchmark. Specifically, the table shows the average number of instructions, program memory accesses and program DRAM accesses (L2 data cache misses) that occur between TLB misses. The number of instructions issued between TLB misses varies from tens of thousands for compute-intensive workloads to hundreds, for data-intensive workloads. For SPECjbb2005, less than five DRAM accesses are made between TLB misses. For data-intensive workloads that may be memory bottlenecked, the DRAM accesses related to page walks are significant.

Table 4.4 compares the behavior of the different MMU caches. For each kind of MMU cache the table shows how many times the MMU cache, the L2 data cache and DRAM

Workload	Ins/Walk	Mem/Walk	DRAM accesses/Walk		
			2MB	1MB	512KB
bwaves	3637.7	2183.8	102.6	104.4	106.2
gamess	37927.8	16905.0	1.1	1.1	1.1
milc	202.3	83.1	3.9	3.9	4.0
zeusmp	3105.2	562.0	77.0	77.8	79.0
gromacs	25399.1	12025.0	42.2	55.4	69.5
cactus	3916.9	2919.4	28.7	30.2	31.9
leslie3d	4185.2	1679.8	67.5	70.5	72.0
namd	49024.9	18498.7	9.0	12.3	16.2
deal	29235.3	10046.7	12.3	14.5	16.9
povray	38328.8	19498.9	1.9	1.9	1.9
Gems	50817.5	19447.8	1.4	1.4	1.4
tonto	30414.8	13711.6	4.5	7.2	28.8
lbm	1844.5	908.1	97.5	101.7	106.7
sphinx3	1858.2	574.4	26.3	28.6	30.2
(avg)	19992.7	8503.2	34.0	36.5	40.4
specjbb	351.0	162.0	2.4	3.4	4.8
Sweep3d	6098.6	3161.3	81.9	83.9	85.4

Table 4.3 : The frequency of TLB misses for each workload, shown as the number of instructions, memory accesses and DRAM accesses between TLB misses. These results are shown for three different L2 data cache sizes and the TLB configuration described in Section 4.5.

are accessed per TLB miss under each of the benchmarks. All caches are using a least-recently-used replacement policy. In these simulations, the unified and path caches have 24 entries and the split caches have  $3 \times 24$  entries. While there are some outliers, most of the applications exhibit similar behavior.

As a baseline, Table 4.4 also presents results with no MMU cache. As expected, with no cache there are four memory accesses per walk. Interestingly, there are only 0.15 DRAM accesses per walk for SPEC CFP2006, meaning that there is a 96% hit rate for page table entries in the L2 data cache. This number varies from application to application, but it never drops below 90%. This demonstrates that page table access for these applications has very high reuse.

Adding any MMU cache drops the average number of memory hierarchy accesses per walk from 4.00 to no more than 1.13 ( $0.99+0.14$ ) for SPEC CFP2006. Note that DRAM accesses only decrease approximately 7%, from 0.15 to 0.14. This means that most of the avoided memory accesses come out of L2 hits, not DRAM accesses. The decrease in TLB miss latency from a MMU cache with these applications comes from the decreased access latency of an MMU cache as compared to the L2 data cache, not an improved hit rate. For Sweep3d, there is so much locality in virtual address use that memory accesses per TLB miss is further reduced to 1.07.

Since these caches do not store L1 page table entries, this result is very close to the minimum number of memory accesses per TLB miss of 1.00. On these applications, all the caches have similar hit rates. In nearly all TLB misses, all the MMU caches can provide the physical page number of the L1 page table page without having to do any memory accesses. From here, the L2 cache still provides most of the L1 page table entries at a hit rate of 88% (only 0.14 of the 1.13 memory accesses are DRAM accesses). These hits come from the fact that the L2 cache is much larger than the MMU caches and can store eight

page table entries in a single cache line.

One benchmark, `tonto`, has more DRAM accesses when an MMU cache is used. However, the rounding in Table 4.4 exaggerates this effect. The actual increase in DRAM accesses is only 0.001 per TLB miss. The MMU cache changes the access pattern to the L2 data cache, so page table entries may be replaced sooner than they would be without an MMU cache, slightly increasing DRAM accesses.

`SPECjbb` has low address locality at a page and cache line granularity compared to other small memory applications leading to high TLB and cache miss rates. However, there is still enough locality in upper level bits of the address to allow reasonably high MMU cache hit rates. On this workload, 2.87 of the three upper level page table entries are still served by the MMU cache, leaving 0.99 and 0.14 accesses for the L2 data cache and DRAM respectively. These 0.14 DRAM accesses per TLB miss are significant compared to the 3.4 DRAM accesses between TLB misses that come from program execution itself.

Since these caches all provide high hit rates, their primary difference is in the number of accesses to the cache required per walk. Since the translation and path cache search for L2 entries first, they are typically only accessed one time per TLB miss. This reduces both latency and power consumption. The page table caches are accessed an average of three times per walk, since they must walk down the page table. If the size of the virtual address space is expanded by adding an additional level, this penalty will increase.

## **Large Memory Applications**

In contrast to the results for small memory applications, the different MMU caches have substantially different hit rates for applications with random access patterns over gigabytes of virtual memory, such as an in-memory database hash join or HPCC `RandomAccess`. For these applications, the reuse of lower level page table entries is low, and there are many

upper level page table entries in use. Consequently, caching as many L3 entries as possible, each of which covers a 1GB region of virtual memory, is critical, but caching L2 entries is of little or no benefit. For a unified cache, a TLB miss that hits on an L3 entry but not an L2 entry will load a new L2 entry into the cache. With LRU replacement, the cache will have about the same number of L2 and L3 entries. The TPC also keeps track of an L2 entry for each L3 entry stored, but it does so in the same associative entry. Therefore, for such applications, a TPC of  $n$  paths is roughly equivalent to a UPTC of  $2 \cdot n$  entries. When the TPC becomes large, there is the possibility that different stored paths will contain the same L3 entry, reducing hit rate. The split caches do not exhibit this effect, so once the cache is large enough to hold all L3 entries in use, there is a 100% L3 hit rate.

A database hash join running over a 16GB region (Figure 4.11) demonstrates this scaling. Here, to have a 50% hit rate on the L3 page table, a 22 entry unified cache is required, while only an 11 path TPC, or a  $3 \times 11$  entry STC is required.

An application using many terabytes of virtual memory would only have high reuse on L4 page table entries, causing 2/3 of unified cache entries to be wasted, instead of just 1/2 for the 16GB application. This is because for every L4 entry stored, an L3 and an L2 entry are also stored, which are effectively wasted. This causes the TPC to be equivalent to a unified cache of three times the size for such workloads.

#### 4.6.2 Sizing considerations

Appropriate sizing of caches is critical for hit rate in many applications. The ability to not store levels that are skipped when a translation cache is used allows such caches to be smaller for a given hit rate. Additionally, the fixed allocation of entries for each level in the split cache designs demands that all levels be large to properly adapt to differing workloads. The TPC and unified caches dynamically allocate entries and adapt well.

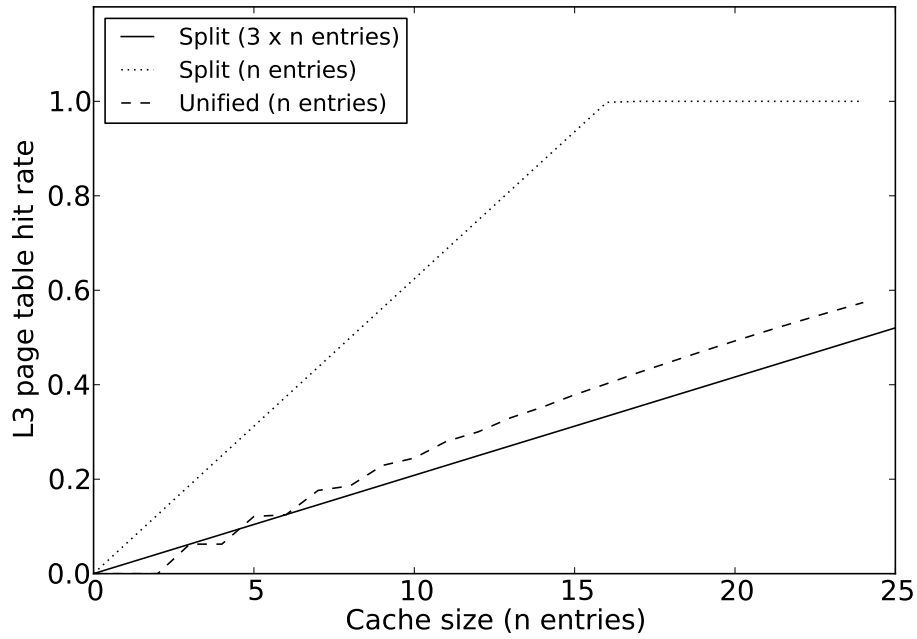


Figure 4.11 : Hit rate compared for the caches with the database join simulation using a 16GB hash table. The different split and unified designs have equivalent coverages.

ASCI Sweep3d operates on a set of different memory regions. When the first is processed, it moves to the next, and so on. After processing the last region, the program wraps around to the first, and the cycle repeats. If the cache is not large enough to hold all the regions, entries corresponding to earlier regions are pushed out before they are used again, and hit rate is very poor (Figure 4.12). Since upper levels of the page table are skipped in the translation cache, the UTC can be slightly smaller than the UPTC.

For the small memory applications, there are relatively few upper level page table entries that are in use. As a result, the hit rate of the unified caches holding  $n$  entries only slightly trails that of the split caches, which hold a total of  $3 \times n$  entries. In this example (Figure 4.13), a unified page table cache holding 23 entries is equivalent to a split cache holding  $3 \times 19 = 57$  entries. Only four entries from the unified cache are stored in upper

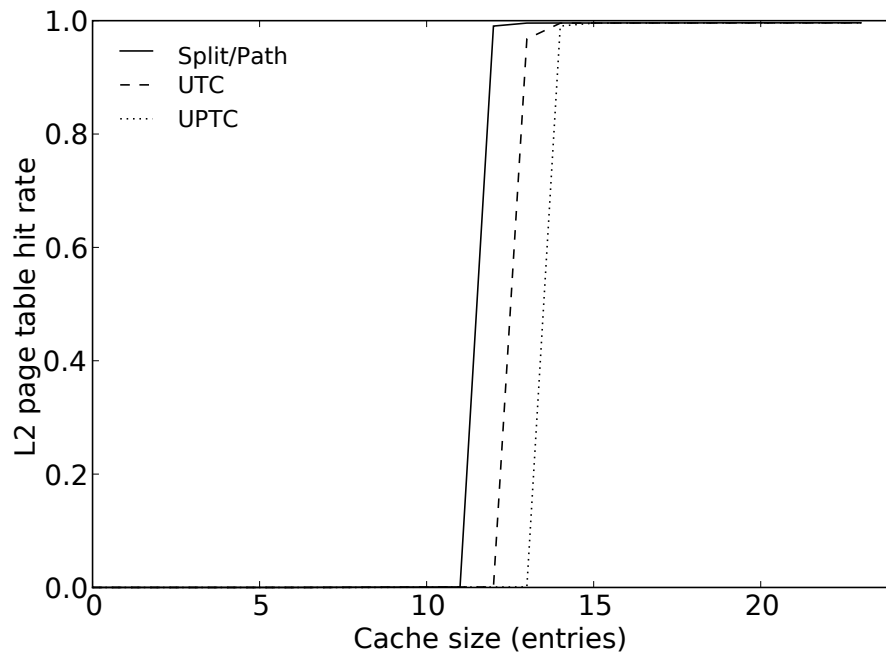


Figure 4.12 : Hit rate for the L2 table for ASCII Sweep3D. This application is very sensitive to sizing. Note that size represents per-level size (all three levels hold  $n$  entries) for the split cache included in Figures 4.12-4.14.

levels of the split caches. These entries are combined with lower level entries in the TPC, allowing a 19 path TPC to be equivalent to the 23 entry unified cache.

While these applications use more lower level entries than upper level entries, having small upper-level caches in a split cache dramatically reduces hit rate for large applications. If the split cache is reduced in size to  $3 \times 8$  entries to match the total size of the unified cache, the L3 table hit rate in the database benchmark is reduced from 99% to 44%. Skewing the distribution of entries from higher levels to lower levels will further impact hit rate. Therefore, it is imperative that all levels of a split cache be large, resulting in considerable area overhead.

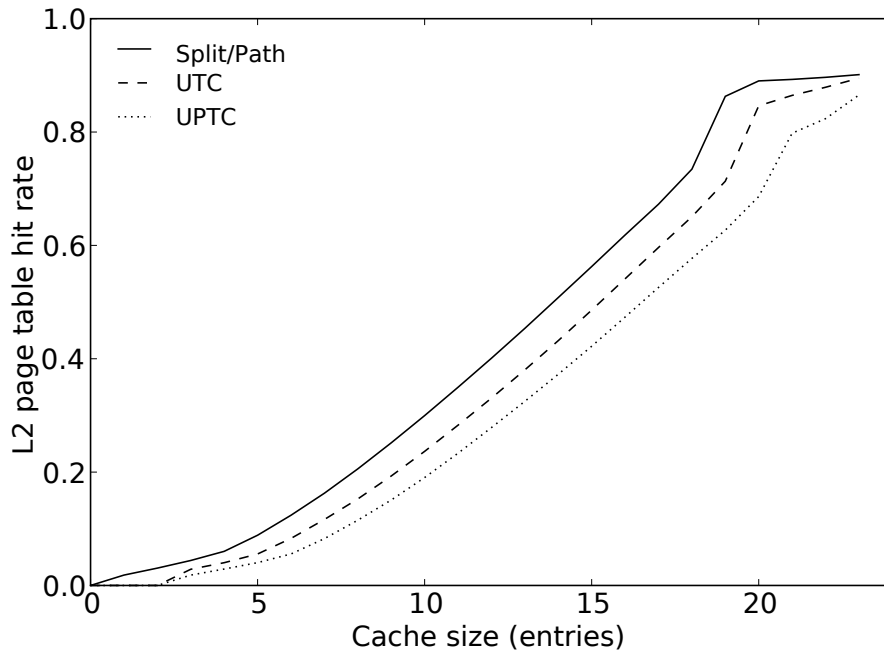


Figure 4.13 : Hit rate for the L2 table alone for the CactusADM component of SPEC CPU2006.

### 4.6.3 Replacement policy

In the unified caches, entries of high reuse (upper level entries) are mixed with entries of lower reuse (lower level entries). This causes the cache to be relatively sensitive to the LRU replacement policy which ensures that frequently accessed components (L4 and L3 entries) are not evicted. In the path and translation caches, these upper level entries are skipped, so they do not need to be protected.

Table 4.6.3 shows the impact of using a random replacement scheme on the number of MMU cache misses (measured by the number of required memory accesses to locate upper level page table entries). While all structures have a lower hit rate using a random replacement scheme, the unified designs are much more reliant on an LRU replacement scheme.



The primary problem with the unified cache designs for the large applications is that entries with high reuse are evicted to make room for entries of low reuse. For example, in the database join the LRU unified caches hold a relatively useless L2 entry for every useful L3 entry in the cache. If a content-aware replacement scheme is used, this problem can be significantly reduced. If the Greedy Dual algorithm is implemented in the UTC instead of using LRU, the size required for a 90% L3 entry hit rate in the 16GB database join is reduced from 52 to 30 entries.

Using our modified LRU replacement scheme (described in Section 4.3.2) with fixed insertion points, this algorithm actually has a higher hit rate than Greedy Dual for the database benchmark. Only 23 entries are required for 90% hit rate. However, the fixed insertion point for L2 entries reduces hit rate significantly for some other applications, such as Sweep3d. The L2 entries used by Sweep3d, if they are inserted near the least recently used position, are replaced before they are reused resulting in a near zero hit rate.

Using the variable insertion-point LRU scheme solves this. For SPEC CFP2006, SPEC JBB2005 and Sweep3d, VI-LRU has a hit rate that is equal to or slightly greater than standard LRU. For the database join, where there are many L2 entries to cache, VI-LRU adapts far better than other replacement schemes. Only 16 entries are required for 90% hit rate in the join benchmark, as opposed to 52 for standard LRU (Figure 4.14). The VI-LRU cache nearly eliminates the conflict between levels seen in the unified cache, and an  $n$  entry VI-LRU UTC has nearly the same hit rate as a  $3 \times n$  entry split translation cache.

## 4.7 Virtualization

While this chapter focuses on the behavior of MMU caches with native execution, they are important to virtualization and nested paging as well. The increased overhead of virtual memory under virtualization compounds the performance impact of an MMU cache. In

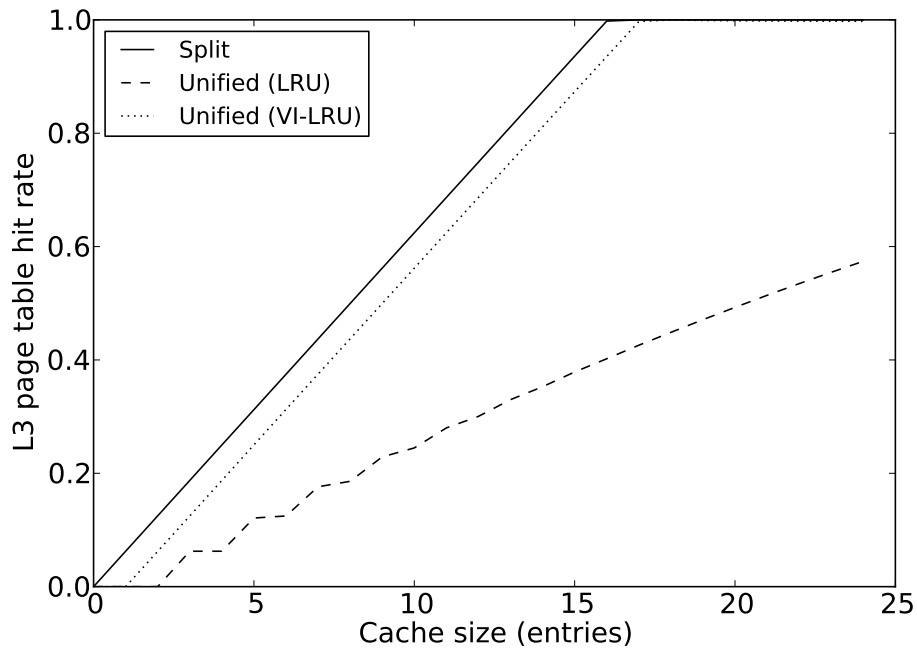


Figure 4.14 : An  $n$  entry unified translation-cache with VI-LRU has nearly the same hit rate as a  $3 \times n$  split cache.

this section, we discuss previous support for virtualization and some possible future work.

In a virtualized system using nested paging, both the guest virtual machine and the underlying virtual machine monitor have their own page table. In effect, the virtual machine monitor's page table is used to create a private guest physical address space for the virtual machine. Thus, the guest's page table is used to translate from virtual addresses to guest physical addresses, and the virtual machine monitor's page table is used to translate from guest physical addresses to host physical addresses. Nested paging with radix tree-based page tables leads to a two-dimensional page table walk because every access to the guest's page table during a page walk may result in a page walk on the virtual machine monitor's page table.

Bhargava *et al.* showed that an extended version of AMD's Page Walk Cache could

effectively cache most of the upper level page table entries in both the guest’s page table and the virtual machine monitor’s page table [1]. In addition, they proposed the introduction of a *Nested TLB* (NTLB) that caches guest physical to host physical translations. In effect, a hit in the NTLB allows the two-dimensional page walk to *skip* the page walk on the virtual machine monitor’s page table. Thus, if every access to the guest’s page table hits in the NTLB, then the number of accesses to the Page Walk Cache and the memory hierarchy is the same as it would be for native execution.

The translation and path caches presented in this thesis could also be extended to support nested paging. Moreover, the NTLB is not inextricably tied to the Page Walk Cache or page table caches in general. A NTLB could be beneficially combined with translation and path caches. A NTLB hit allows the two-dimensional page walk to skip the entire page walk on the virtual machine monitor’s page table for a single guest page table access, but not the accesses to the upper levels of the guest’s page table. This requires a translation or path cache. Moreover, a translation or path cache could accelerate page walks on the virtual machine monitor’s page table when a NTLB miss occurs.

## 4.8 Conclusions

Since the x86 architecture began using a radix tree page table for address translation in the 80386, the depth of the page table has increased by one level with each passing decade. Unfortunately, without an MMU cache, the page table walk for address translation requires a memory reference for each level of the radix tree. Therefore, MMU caches have become critical components of current and future x86 processors. This chapter has presented a quantitative and qualitative comparison of the design space of such MMU caches, including three new designs. While this thesis focuses on x86 processors, the results should apply generally to any architecture that uses a radix tree page table.

While AMD and Intel have both developed MMU caches for their microprocessors, this chapter has introduced a unified translation cache with a modified LRU replacement scheme that is superior to both existing devices. It adapts well to varying workloads, unlike a split translation cache, as implemented in Intel's Paging Structure Cache. It also prevents conflict between entries of low and high reuse, unlike the LRU unified page table cache, as implemented in AMD's Page Walk Cache.

	No Cache			UPTC (24 entry)			SPTC (3x24 entry)			UTC (24 entry)			STC (3x24 entry)			TPC (24 path)		
	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM	S	L2	DRAM
bwaves	-	3.72	0.28	2.98	0.82	0.25	2.99	0.80	0.24	1.06	0.81	0.26	1.03	0.80	0.24	1.03	0.80	0.23
gamess	-	3.95	0.05	2.94	1.13	0.05	2.94	1.13	0.05	1.15	1.13	0.05	1.15	1.13	0.05	1.15	1.13	0.05
milc	-	3.90	0.10	3.00	0.91	0.10	3.00	0.91	0.09	1.01	0.91	0.09	1.00	0.91	0.09	1.00	0.91	0.09
zeusmp	-	3.81	0.19	2.99	0.85	0.17	2.99	0.85	0.17	1.02	0.85	0.17	1.02	0.85	0.17	1.02	0.85	0.17
gromacs	-	3.80	0.20	2.96	0.91	0.20	2.96	0.91	0.20	1.08	0.90	0.20	1.08	0.91	0.20	1.08	0.91	0.20
cactus	-	3.84	0.16	2.96	1.12	0.16	2.99	0.98	0.15	1.23	1.10	0.15	1.13	0.98	0.15	1.13	0.99	0.15
leslie3d	-	3.79	0.21	2.98	0.88	0.20	2.99	0.85	0.19	1.06	0.88	0.20	1.04	0.85	0.19	1.04	0.86	0.19
namd	-	3.86	0.14	2.88	1.15	0.13	2.88	1.14	0.13	1.23	1.15	0.13	1.23	1.14	0.13	1.23	1.14	0.13
deal	-	3.89	0.11	2.93	1.07	0.11	2.93	1.07	0.11	1.15	1.07	0.11	1.15	1.07	0.11	1.15	1.07	0.11
povray	-	3.92	0.08	2.92	1.10	0.08	2.92	1.10	0.08	1.15	1.10	0.08	1.14	1.10	0.08	1.14	1.10	0.08
Gems	-	3.93	0.07	2.89	1.18	0.07	2.89	1.18	0.07	1.20	1.18	0.07	1.20	1.18	0.07	1.20	1.18	0.07
tonto	-	3.93	0.07	2.94	1.07	0.08	2.94	1.07	0.08	1.12	1.07	0.08	1.12	1.07	0.08	1.12	1.07	0.08
lbm	-	3.79	0.21	2.99	0.83	0.19	2.99	0.83	0.18	1.02	0.83	0.19	1.01	0.83	0.18	1.01	0.83	0.18
sphinx3	-	3.76	0.24	2.99	0.80	0.23	2.99	0.79	0.23	1.02	0.80	0.23	1.02	0.79	0.23	1.02	0.79	0.23
(avg)	-	3.85	0.15	2.95	0.99	0.14	2.96	0.97	0.14	1.11	0.98	0.14	1.09	0.97	0.14	1.09	0.97	0.14
specjbb	-	3.83	0.17	2.98	0.97	0.17	3.00	0.93	0.17	1.14	0.97	0.17	1.11	0.93	0.17	1.11	0.95	0.17
Sweep3d	-	3.77	0.23	2.95	1.01	0.22	2.98	0.87	0.20	1.13	0.94	0.21	1.07	0.87	0.20	1.07	0.88	0.20

Table 4.4 : The number of caching structure accesses (S), L2 data cache hits (L2), and DRAM accesses (DRAM) per TLB miss for the various LRU cache designs over the SPEC CFP2006, SPECjbb2005 and sweep3d benchmarks.

Cache	LRU	Random	Increase
UPTC	0.61	1.00	63%
UTC	0.53	0.79	49%
TPC	0.51	0.65	28%
SPTC	0.51	0.64	25%
STC	0.51	0.63	23%

Table 4.5 : The average number of misses per walk for random and LRU replacement, normalized to Random UPTC (lower is better), and the relative increase in misses using random replacement over LRU replacement.

## CHAPTER 5

---

### Alternate page table formats

---

The MMU cache is designed to reduce the overhead from having to access the upper levels of the page table. Ideally, this would allow a single access to the page table per TLB miss. However, these accesses are a result of the specific page table format used, and replacing the page table itself could potentially reduce the number of accesses per walk required to a similar number.

We compared the memory access behavior of the cached radix tree page table with its biggest rivals, hash-table based Inverted Page Tables and direct-mapped Translation Storage Buffers. These structures are attractive since they contain only one level, and are therefore insensitive to address space size. However, the unavoidable presence of hash and structural collisions, low access locality and their inability to handle multiple page sizes efficiently cause them to require far more memory accesses than a cached radix table.

#### 5.1 Hashed page tables

We have shown that MMU caches can significantly reduce the overhead of using a radix tree page table, however the possibility remains that the radix tree page table itself should be replaced. The traditional competitor to the radix tree page table is the inverted page table, which uses a hash table to store a large and sparsely used address space efficiently [11]. These designs are usually seen as superior to a multi-level table, because they only need

to be referenced once, whereas the radix table requires one access per level. However, hash collisions are unavoidable, so many accesses may require more than one reference to follow a collision chain.

Additionally, we have shown in this thesis that MMU caches can reduce the number of memory accesses per walk to nearly one as well. To compare the cached radix table against an inverted page table, a simulator was constructed that maintains and references a hash table storing all the memory locations used during a process' lifetime. The hash table used models that used by the Intel Itanium [31], though our implementation differs. The Itanium handles translations in hardware only if the desired translation is at the front of the collision chain. Otherwise, an interrupt occurs and the collision chain walk is done in software. Our simulation performs the entire process in hardware, and is therefore an optimistic design. The number of accesses to this table were counted, as well as the number of such accesses that hit in the L2 data cache.

When the hash table contains twice as many buckets as there are pages to store, the hash table walker references approximately 1.2 locations per TLB miss, *regardless of benchmark or access pattern*. This number comes from the average length of a collision chain, which is a function only of the fullness of the hash table if a sufficiently uniform hash function is used [30]. This number compares poorly to the average number of L2 and memory accesses required per walk of the SPEC CFP2006 applications using page table caching of 1.13. While the hashed page table is insensitive to address space size, it is unable to take advantage of the great locality seen in virtual address space usage like MMU caches can.

Compounding this issue is the fact that references into the hash table show no spatial locality. Whereas consecutive pages in virtual memory are usually mapped by consecutive entries in the radix table, they are not usually mapped by consecutive entries in a hash table. Since there is usually locality in the access pattern of L1 page table entries in a radix



	Uncached		TPC	
	4KB	2MB	4KB	2MB
Page Size				
L2 Hits	2.90	2.92	1.11	1.15
DRAM	1.10	0.06	1.09	0.06

	IPT (1)		IPT (2)		IPT (16)	
	4KB	2MB	4KB	2MB	4KB	2MB
Page Size						
L2 Hits	0.01	0.00	1.16	1.16	0.55	0.54
DRAM	1.29	1.29	1.14	1.14	1.48	1.49

Table 5.1 : L2 hits and DRAM accesses to the page table per walk for a radix tree page table for the 16GB database join benchmark. Results are shown for an uncached radix tree, a cached radix tree, and a half-full inverted page table with various numbers of clustered translations per tag.

table, these entries are much more effectively cached by the L2 data cache than the entries of the hashed table are. For the SPEC CFP2006 applications examined earlier, only 44% of the 1.2 accesses/walk are served by the L2 data cache. Overall, the inverted page table increases the number of DRAM accesses per walk by *over 400%*.

Spatial locality can be increased by storing multiple adjacent translations with a single tag, as used in *clustered page tables* [7]. This technique also reduces the overhead (virtual address tag and chain pointer) for the hash table. However, for this technique to be effective each virtual tag must be associated with many translations. This means that some translations will need to load multiple cache lines. Additionally, the frequency of hash collisions is not reduced over a standard inverted page table.

Even if the virtual address space is used without locality, as in the database join, the radix tree page table still requires fewer DRAM accesses than a hashed page table. Table 5.1 shows the memory use per TLB miss for a join using a 16GB hash table. With 4KB pages, the radix tree page table requires fewer DRAM accesses/walk than the inverted page table until a 48GB inverted page table is used.

For this application, clustering does improve L2 cache hit rate since the page table is smaller. However, the tag and data often lie in different cache lines, which requires an increase in the total overall memory references required to perform a translation. Matching the total cluster size to the size of a cache line improves this. However since cache line size may change from implementation to implementation, the appropriate cluster size may change as well.

For larger applications such as this, large page support becomes important. When it comes to supporting the simultaneous use of multiple page sizes, radix trees have an advantage over inverted page tables. With the radix tree, if 2MB pages are used for mapping most of the virtual address space, the entire page table can be cached in the L2 data cache, because the 2MB page mapping takes the place of an L2 entry in the page table and eliminates the need for an entire L1 page table page (see Figure 2.2). This reduces the number of DRAM accesses per walk dramatically for the radix table designs and also the number of overall memory hierarchy accesses to below that seen in the inverted page table, as shown in Table 5.1.

In contrast, the simultaneous use of large and small pages does not reduce the size of an inverted page table, and so its memory accesses do not change. In essence, the hash function must take into account the size of the virtual page, but it cannot know the page's size *a priori* if multiple page sizes are in use. Consequently, for a large page, the inverted page table must still have a page table entry corresponding to each of the small pages that make up the large page. Each of these page table entries will, however, designate the mapping as part of a large page, and the TLB will be loaded with a single large page mapping.

## 5.2 Translation Storage Buffers

The SPARC architecture has traditionally handled TLB misses in software. To accelerate TLB misses, the processor supports a software-managed, direct-mapped cache of translations called the *Translation Storage Buffer* [32]. On a TLB miss, the CPU derives an index from the lower order bits in the virtual address and checks to see if a corresponding entry is present in the TSB. Although earlier processors performed this TSB lookup in software, some current processors implement it in hardware. Like the inverted page table, a TSB entry stores a tag (a virtual page number) and a translation (a physical page number). Unlike the inverted-page table, there is no chaining. If a translation is not present in the TSB, a software fault occurs.

To compare this design to the radix-tree design, a TSB simulator was also developed. Like the inverted page table simulator, the TSB simulator counts L2 and DRAM accesses per TLB miss. In addition, the TSB simulator also counts software faults that occur when a translation is not present in the cache. Traces are simulated in a two-pass manner. The first pass populates the cache with the translations present in the trace. The second pass actually simulates accesses to the cache, counting hits and misses. This ensures that only conflict misses are counted, providing a lower-bound for cache misses.

Our results show that the TSB uses the L2 data cache poorly as compared to the radix-tree. For example, the `zeusmp` component of the SPEC CFP benchmark generates 0.058 DRAM accesses per TLB miss using the radix-tree with an MMU cache whereas the TSB required 0.078 DRAM accesses per TLB miss. This increase is due solely to the larger size of a TSB entry as compared to a radix tree page table entry. The TSB entry contains a tag and data, whereas the radix-tree only needs to hold data. This increased size reduces the number of entries that will fit in a single L2 cache line from eight to four.

In addition, the TSB also generates 0.024 software faults per TLB miss using the current architectural maximum size of 1 megabyte. These are likely to be extremely expensive, generating not only data cache misses, but also instruction cache misses. While increasing the size of the L2 data cache would reduce the number of L2 cache misses, it would not reduce the number of software faults.

The PowerPC architecture uses a page table that is somewhat of a cross between the Itanium style hashed page table and SPARC's TSB [33]. PowerPC hashes virtual addresses like Itanium does. However, instead of using a collision chain, it maintains a fixed size array for collision resolution. Software must handle the translation if none of the entries in the array match the virtual address, much like the TSB. The overall space overhead of such a system is likely to be even higher than other hashed page table formats since the size of this array is fixed even if not all entries are filled. However, accesses to the collision chain are more likely to be cached since they are close together in memory and may reside in the same cache line.

### 5.3 Conclusion

MMU caches dramatically change the trade-offs in page table design for large address spaces. Radix tree page tables make more effective use of the processor's L2 cache than either inverted page tables or translation storage buffers. Radix tree page tables have a smaller page table entry size, because both inverted page tables and translation storage buffers must include a tag in the page table entry. Thus, the L2 cache is able to hold more page table entries from the radix tree, increasing its coverage and reducing DRAM accesses. So, while these alternate structures are superior to the radix tree page table on its own for large address spaces, a well designed MMU cache renders the radix tree organization far superior.

## CHAPTER 6

---

### SpecTLB: Parallelizing TLB Miss Handling

---

The use of large pages reduces the performance overhead of virtual memory by increasing TLB coverage. Each entry in the TLB covers a larger region of virtual memory, so the entire TLB is able to translate a larger region of the address space without walking the page table. However, this increased coverage does not come for free.

The page size is the minimum unit of a program's address space that the operating system is able to allocate and protect. The operating system must allocate an entire page of physical memory, regardless of how much of that space will be used by the application. This can lead to excessive physical memory use if a program fragments its virtual memory use. Additionally, application permissions (read/write/execute) must be consistent across an entire page. Finally, the operating system can only tell if memory has been changed or accessed on a page granularity. This means that if large files are used for memory-mapped files, the operating system must write an entire large page worth of data to disk, regardless of how much of the original memory was modified.

On x86-64, page sizes are only available in three sizes: 4KB, 2MB and 1GB. These sizes correspond to the size of virtual address space represented by an L4, L3 and L2 page table entry, respectively. The large gap between these sizes makes page size selection critical. Using a 2MB page increases L1 TLB coverage by a factor of 512 compared to using small pages, but it can also increase physical memory requirements for a sparsely

utilized address space by the same factor. The proper page size for a region of memory may change from application to application or even from execution to execution. Therefore, some modern operating systems take an automatic approach to selecting page size.

FreeBSD's *reservation based memory allocator* uses small pages for all memory by default [5]. After a program uses every small page within an entire 2MB region of virtual memory, that region is *promoted* to a large page. To prepare for this, the operating system places small pages it thinks are likely to be promoted into large page reservations. In a reservation, 4KB pages are aligned within a 2MB region of physical memory corresponding to their alignment within their 2MB region of virtual memory. This means that within a reservation, consecutive virtual pages will also be consecutive physical pages.

This contiguity and alignment can be exploited by the MMU to predict the physical address of pages that miss in the TLB by interpolating from nearby pages. In this chapter, we present the *SpecTLB*, a novel TLB-like structure that provides speculative translation for small pages that are part of a large page reservation. While the underlying page table still must be walked to verify the speculative translation, this TLB walk is done concurrently with the speculative memory access and execution. This new capability allows the operating system to maintain fine-grained protection and allocation over memory while eliminating the latency of the resulting TLB misses.

We show that the SpecTLB is able to eliminate the latency penalty from a majority of TLB misses with an unmodified version of FreeBSD. However, one of the key contributions of the SpecTLB is its ability to achieve large-page like performance when large pages are impractical to use, such as in virtualization. Traditional hypervisors implement I/O by marking pages of the guest physical address space that contain memory mapped I/O as unavailable. When the guest system accesses them, the hypervisor is invoked which emulates the I/O device. Ideally, the physical memory space of a virtualized guest would

be stored as a small number of 1GB pages. However, the hypervisor cannot control which guest physical pages the guest operating system will use for I/O. Therefore, the hypervisor must have fine-grained protection control. With a speculative TLB, this space can be stored in a 1GB reservation with both data and I/O pages mixed together. All accesses are made speculatively, as if they were to a data page. Therefore, all guest physical memory accesses can proceed without blocking for a TLB miss. If the address turns out to be part of a data region, the speculative work is committed. However, if an address is part of an I/O region, the speculative execution will not be committed, and the hypervisor will be invoked as before.

This chapter is organized as follows. Section 6.1 discusses the operation of the reservation-based memory manager. Section 6.2 discusses the design of the SpecTLB. Section 6.3 and 6.4 discuss our simulator and simulation results. Section 6.5 discusses proposed software extensions to support the SpecTLB. Finally, we conclude in section 6.6.

## 6.1 Background

The ability to predict the physical address of operations that miss in the TLB is dependent on a reservation based memory allocator, first suggested by Talluri and Hill [6]. Navarro *et al.* ([5]) extended this idea to a practical memory allocation system and implemented it under FreeBSD. This extended design reclaims underfilled reservations, allowing the empty pages to be used by other processes.

### 6.1.1 FreeBSD Reservation System

When a process allocates virtual memory, through `mmap()` or indirectly through `malloc()`, the operating system maintains metadata about what operation allocated that space. At this point, however, physical memory is not typically allocated, nor the page table

updated until the process tries to access a location within that virtual allocation. Then, the memory management unit invokes the fault handler, which locates the metadata associated with the faulting virtual address.

The fault handler uses that information to predict if the virtual memory space used is likely to be contiguous and larger than a superpage. For example, a memory mapped 5KB file will not use an entire superpage, so it will not be placed in a reservation. If the handler decides that a superpage is appropriate, it will reserve an entire 2MB physical page, and assign the 4KB page within it that is virtually and physically aligned to the faulting virtual address. When the program faults again on the next accessed page, the fault handler will recognize that the address is part of a reservation, and it will again return the virtually and physically aligned page within that reservation. When all blocks in the reservation are filled, the page is promoted to a superpage. However, under memory pressure, this reservation may be broken down if it is never filled. The virtual memory system can then return the unused pages back to the free pool of small pages.

## 6.2 Page table speculation

The SpecTLB is a translation-lookaside buffer that tracks underfilled large-page *reservations* instead of large pages themselves. On a TLB miss, the SpecTLB is consulted to see if the faulting virtual page may be part of a large page reservation. If so, the physical page number of the faulting page can be interpolated from the physical page number of the reservation and the small page's position within the large reservation.

The primary difference between the SpecTLB and a traditional TLB is that mappings generated by the SpecTLB are predictions; they are not guaranteed to be correct. An entry in the SpecTLB indicates that the operating system has placed small pages within a particular physical reservation in the past, but it does not guarantee that any particular virtual



page was placed in the reservation or even that the page is valid. This distinction means that a TLB miss that does hit in the SpecTLB must still be validated against the underlying page table. However, the interpolated translation can be used for speculative execution while the page walk itself continues in parallel. If the interpolated translation matches the result from the page walk, the speculative work can be committed and execution continues. If the results differ, the speculative work is cancelled and execution restarts from the first incorrect prediction. While the SpecTLB does not reduce the overall memory bandwidth required by the MMU, it does reduce the latency penalty from TLB misses by removing the page walk from the critical path of execution.

This relaxed requirement of correctness allows two different variants of the SpecTLB to be built: one that requires software support and one that does not. Both are presented in detail here with the software-independent variant simulated in later sections.

### **6.2.1 Explicit page table marking**

The SpecTLB maintains a set of large page reservations it believes the operating system is assigning to the current process. This set is maintained by monitoring which small page table entries are marked by the operating system as being part of a large page reservation. These marks are implemented using one of the currently unused bits in bottom level (L1) page table entries in x64-64. They do not effect how these entries are translated; they are still standard small pages and they are only used to maintain the contents of the SpecTLB.

For purposes of describing the operation of the SpecTLB, we will use a similar simplified address representation as described in Section 2.1. Virtual addresses are split up into four indices and an offset: e.g. (0b9, 00c, 0ae, 0c2, a2e). Physical addresses are similarly divided into nine-bit parts of a physical page number and a page offset: e.g.

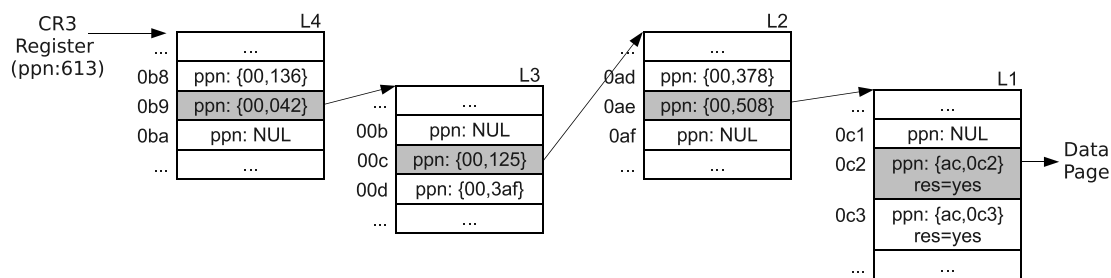


Figure 6.1 : An example of a page table containing a marked reservation.

{0ac, 0c2, a2e}. For simplicity, only eighteen bits (two parts) of the 40-bit physical page number are shown in diagrams.

Figure 6.1 shows the page table walk for the translation from (0b9, 00c, 0ae, 0c2, a2e) to {0ac, 0c2, a2e}. This entry is marked by the operating system as being part of a reservation, so it is added to the SpecTLB (Figure 6.2):

Virtual Page Number	Physical Page Number
(0b9, 00c, 0ae)	{0ac}
...	...

Figure 6.2 : An example of the contents of a the SpecTLB.

Note that only the upper bits of the translation, those that select a particular large physical or virtual page are stored. A SpecTLB entry for a particular address is effectively whatever the standard TLB entry would be if that address were part of a large page.

On a subsequent TLB miss, the SpecTLB is searched like a normal TLB. If a subsequent TLB miss is for virtual address (0b9, 00c, 0ae, 0c3, 001), the newly added SpecTLB entry will match. This means that the new virtual address is part of that same virtual large page and that the operating system likely placed the corresponding physical page within the physical reservation. The speculative translation concatenates the stored phys-

ical page number of the matching reservation with the large page offset from the virtual address, yielding  $\{0ac, 0c3, 001\}$ .

Of course, this translation may not be valid. The underlying reservation may have been broken down by the operating system or the virtual address may not even be valid. Therefore, while the processor can use this translation, it must do so speculatively. While program execution continues, the standard x86 page walk happens concurrently. If the predicted physical address matches the actual address, the speculative work may be committed. Otherwise, execution must roll back to the point of the misprediction.

Since SpecTLB translations are always confirmed against the underlying architectural page table, consistency is not as important to them as it is to MMU caches and TLBs. Therefore, implementations can be more lazy about SpecTLB invalidation than they can be with MMU caches. Stale entries do not lead to incorrect operation, as they do in an MMU cache or TLB. Invalid translations generated by stale entries will be corrected automatically. Our implementation only invalidates the SpecTLB on a context switch, though even this is not strictly necessary.

### 6.2.2 Heuristic reservation detection

The above description of the SpecTLB requires explicit marking of the page table. This requires both a modification of the x86 page table architecture and the operating system itself. However, it is possible to build a variant of the SpecTLB that detects reservations and requires no modification to system software.

To allow a region of memory to be promoted to a large page, small pages must be aligned within their large page reservation. In the example above, virtual address  $(0b9, 00c, 0ae, 0c2, a2e)$  is mapped to physical address  $\{0ac, 0c2, a2e\}$ . The virtual page's offset within a 2MB virtual page is  $0c2$ , equal to the physical page's off-

set within a 2MB physical page. Specifically, `vaddr[29:20] == paddr[29:20]`. This equality can be used as a heuristic to signal that the operating system has placed this page within a reservation. While it has a false positive rate of 1:512 (assuming 4K pages that are not part of a reservation are placed randomly), it has a zero false negative rate.

The heuristic based SpecTLB uses this detection scheme to maintain its contents. Translations are inserted when `vaddr[29:20] == paddr[29:20]` and entries are removed when they lead to false predictions.

### 6.2.3 Memory side-effects

When a reservation is broken down, the pages that are reclaimed can be reused for any purpose, including for I/O. However, this opens up the possibility that a speculative access will be made to one of these reclaimed pages. On such an access, the processor must guarantee that any operation performed speculatively can be cancelled. For typical reads to cached memory, this is trivial. Operations are simply cancelled and not written back from the reorder buffer. However, if the speculated address is part of an uncachable region of memory or memory-mapped I/O, special care must be taken. Unfortunately, it is not possible to determine if a region of memory is part of a special region *a priori*, since that information is stored in the page table. Typical speculative execution systems avoid this problem because an access is never made to memory without having first been translated through the TLB. This ensures allows the processor to know which memory addresses cause side effects.

Tagging requests throughout the memory system as being speculative should be sufficient to ensure safety. While the exact architectural implementation is beyond the scope of this chapter, this tag could be used to prevent speculative accesses from reaching an I/O controller or writing to memory. Additionally, this will allow new speculative execution

from other systems such as the prefetcher or branch predictor to make similar memory accesses without going through the TLB.

Finally, the explicit marking SpecTLB can avoid these problems entirely by ensuring that uncachable and I/O memory is never mapped into 2MB pages of memory that also contain data pages with the reservation bit set.

## 6.3 Methodology

SpecTLB performance was analyzed using a custom functional simulator executing memory traces generated by a platform simulator. Unlike the analysis of MMU caches performed in Chapter 4, the behavior of the operating system is important to the SpecTLB. In the MMU cache simulations, only virtual addresses affect behavior. However, the mapping of virtual to physical address is critical to simulating the SpecTLB. Therefore, these memory traces need to include both virtual and physical addresses. This precludes the use of the synthetic memory traces used earlier.

### 6.3.1 Platform simulator

The AMD SimNow [23] platform simulator was used to run various benchmarks under FreeBSD 8.0-Release for x86-64. A custom analyzer plugin to SimNow records each virtual memory access made by the simulated system along with the associated physical address and page size. This trace includes all memory loads and stores made by the guest operating system and processes, but it does not include instruction or page table loads. Instruction loads are not modelled by SimNow, so they are not included in this study. Page table loads are simulated by the SpecTLB simulator. TLB invalidations are included in the trace by monitoring the value of the CR3 register, changes on every context switch. Finally, this plugin counts the total number of instructions executed during the trace.

### 6.3.2 Benchmarks

Traces were collected for several popular benchmarks, including the SPEC CPU2006 suite [24], SPECjbb2005 [25], the NASA Advanced Computing Parallel Benchmarks (NAS) suite [34], benchw [35] and an ad-hoc Python microbenchmark. However, not all of the benchmarks in the SPEC CFP2006 suite could be compiled with the standard tool chain in FreeBSD 8.0, so *soplex*, *calculix* and *wrf* are not included in this study. SPECjbb2005 was run on one warehouse, and Sweep3d was run on a 150x150x150 grid. The NAS benchmarks are configured to use a class-C problem size, whenever possible. The benchw benchmarks function similarly to TPC-H. Specifically, this benchmark executes a `JOIN` between two tables of approximately one gigabyte in size under PostgreSQL 8.4 using default tunings. This benchmark was profiled both in a fresh boot configuration and on a second run, where the data tables are cached. The Python benchmark is a custom microbenchmark that initializes a large array of long integers.

### 6.3.3 SpecTLB Simulation

The SpecTLB simulation uses a modified version of the TLB/cache model from Chapter 4. The TLB is modified to include a 128 entry L2 TLB for large pages, as described in [1]. The 64-entry, fully-associative L1 TLB, the 512-entry, 4-way set associative L2 TLB for small pages and the 1MB L2 data cache are unmodified. Again, the processor's L1 data cache is not accessible to the MMU, so it is not simulated.

The simulator reads the memory trace, running each virtual address through the TLB and data cache model. On a TLB miss, the SpecTLB is searched to see if a possible matching reservation can be found. If so, a speculative physical page number is generated. If this page number matches the actual page number stored in the trace, the speculation is

correct. For simplicity, only the bottom level of the page table is simulated, and a dedicated MMU cache is not included in this model. The page table entry is loaded from cache, and PTE cache hit rates are maintained separately for speculated and unspeculated page table accesses.

Since explicit reservation marking is not yet included in any operating system, the heuristic based SpecTLB is implemented. As a baseline, the number of reservations tracked by the SpecTLB is set at 24, the number of entries in the MMU caches explored earlier. This size is varied from one to forty-eight.

## **6.4 Simulation results**

Simulation results show that the SpecTLB can often accurately interpolate the physical address of memory operations that cause TLB misses. This results in the ability to remove a majority of the high-latency DRAM accesses related to memory management from the critical path of execution.

Even with a heuristic-based approach to detecting reservations, mispredictions are rare. Therefore, the power overhead to such speculation is minimal. A SpecTLB only needs to be of moderate size, approaching maximum hit-rate with tens of entries. Finally, unlike MMU caches, hit rate is relatively unaffected by choice of replacement policy.

### **6.4.1 TLB miss parallelization**

When a translation is able to be accurately predicted, all memory and MMU cache accesses required to serve the TLB miss are removed from the critical path of execution. Tables 6.1 and 6.2 show simulation results from testing the SpecTLB design against various benchmarks.

A speculative translation is only attempted when a matching reservation is found. This

occurs at a per-benchmark average rate of 56%, though it varies greatly by specific workload. As shown in tables Tables 6.1 and 6.2, some benchmarks, such as `mcf` find a reservation and attempt a speculative translation over 99% of the time. Others, such as `ep.C` do so less than 1% of the time. The ability to speculatively translate addresses depends on the locality in which a program uses its address space and if it is used in a fragmented manner, leading to underfilled reservations.

Even using heuristic-based reservation detection, prediction accuracy has a per-benchmark average above 99%. While some benchmarks have particularly low prediction accuracies (less than 40% in the case of `dc.B`), speculation happens comparatively rarely in these workloads. Even with the high miss rate, only 5% of TLB misses generate a misprediction in `dc.B`. The low rate of speculation here is likely due to its small working set, which would lead to comparatively few reservations being made. An explicit reservation marking based SpecTLB would eliminate these mispredictions.

When addresses are accurately predicted, the page table walk can be overlapped with speculative execution using the predicted address. A high prediction rate combined with high accuracy allows useful work to be performed in parallel with much of the overhead of virtual memory. This is quantified by counting the number of total L2 data cache misses made by the MMU as well as the fraction that are overlapped through successful prediction. Since DRAM accesses are so slow, this fraction should translate into a proportional speedup of overall TLB miss handling. Tested benchmarks have an average of 53% of their MMU-related DRAM accesses overlapped with speculative execution by successful prediction. The benchmark which sees the largest number of TLB misses per instruction, `mcf`, has 98% of its DRAM accesses overlapped.



### 6.4.2 Overlap opportunity

Superscalar microprocessors can speculatively execute only a limited number of instructions as constrained by the size of their reorder buffer. If the instructions after a TLB miss complete quickly, the reorder buffer may fill before the parallelized page walk completes. The processor then has to stall until the page walk is complete, as it did before the SpecTLB. Fortunately, the load or store which triggered the TLB miss is by definition to an address not recently accessed, or else it would have hit in the TLB. Therefore, it is less likely to be cached and may be a long-latency operation itself. The high-latency TLB miss can be overlapped with the high-latency load or store before the reorder buffer starts to fill.

To examine this, our simulator was instrumented to determine the L2 cache hit rate for those memory operations that cause TLB misses. The per-benchmark average hit rate for these operations is only 47% and is as low as 6% for some workloads (like `leslie3d`). These results are presented for all workloads in Tables 6.1 and 6.2. If these operations are reads, this presents significant latency in a single instruction that can be worked on while the parallelized page walk completes. On the other hand, write operations cannot be committed, so they must wait for the speculative page walk to complete.

### 6.4.3 Power overhead

Speculative architectures often increase power by performing needless work. Since the SpecTLB does not provide a speculative translation when a request is not part of a tracked reservation, misprediction rates are extremely low (Tables 6.1 and 6.2) and little needless work is done. Therefore, when the SpecTLB is unable to provide predicted translations, the power penalty from its use should be low.

The mispredictions that are present are either for pages that are not present, that were

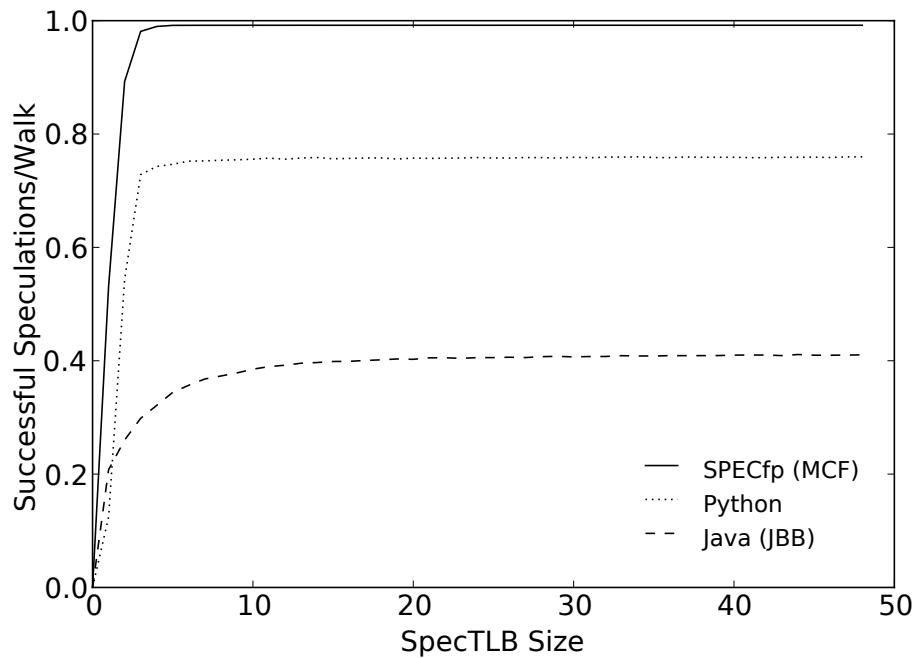


Figure 6.3 : Speculation success rates for different benchmarks for different sizes of SpecTLB. These results are for a random replacement policy.

allocated after a reservation was broken down or were made using a false SpecTLB entry. Incorrect SpecTLB entries are generated in the when a page is aligned by chance, not because it was part of a reservation. Using explicit marking avoids these false entries.

#### 6.4.4 Sizing considerations

Like the MMU cache, even a relatively small SpecTLB is effective. Each entry covers 512 small pages, or 2MB of virtual address space, so a small device leads to reasonably high coverage. Figure 6.3 shows that maximal performance is attained by 24 entries. For SPECjbb2005, the benchmark that requires the most entries, reducing device size from 24 to 12 entries only reduces successful speculation rate by 3%. Other benchmarks are impacted even less.

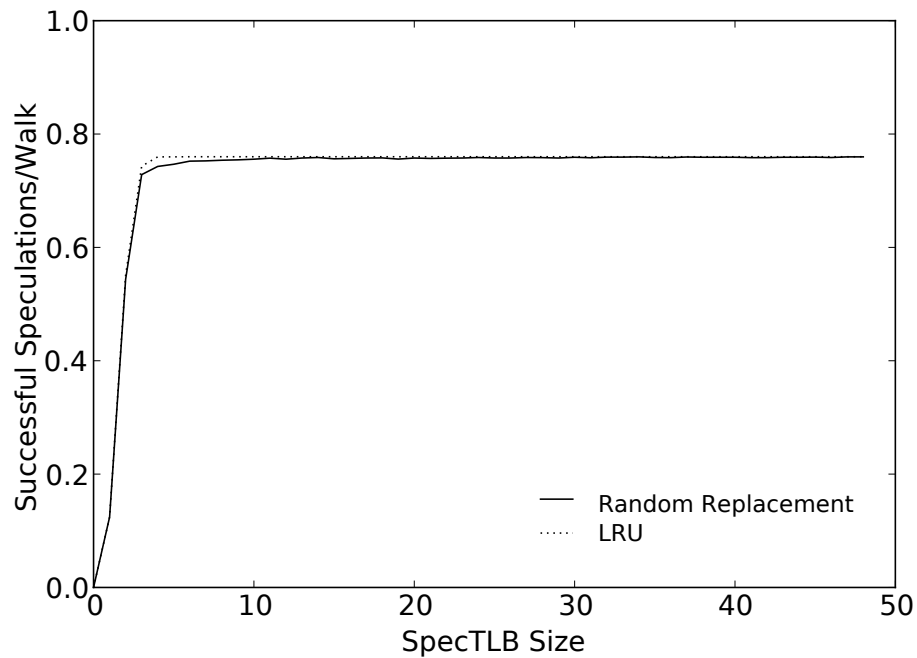


Figure 6.4 : The LRU and random replacement policy are compared for the Python benchmark.

#### 6.4.5 Replacement policy

Unlike MMU caches, only a single entry is accessed in the SpecTLB per translation. This precludes the level conflict seen earlier. Therefore, the SpecTLB is comparatively unaffected by replacement policy. Figure 6.4 shows the successful speculation rate for the Python benchmark for different sized devices. At 24 entries, the LRU cache outperforms the random replacement cache by less than a half a percent, which is unlikely to outweigh its implementation cost. At all sizes, the performance difference is never greater than 2.5% for this benchmark.

## 6.5 Discussion

Operating systems that use reservation based memory allocation are currently designed to use reservations in the hope that those reservations will fill and memory will be promoted to a large page. FreeBSD is tuned to reserve memory only when there is a possibility that it will be able to promote a region to a large page. The results of the previous sections show that even with these tunings, there exist enough underfilled reservations that an accurate prediction can be made in 53% of the workloads tested in Section 6.4. However, if system software is modified to create reservations even when they cannot be promoted, the SpecTLB can allow large-page like performance without promotion. While this thesis is primarily architectural, we discuss several conjectural software modifications here as future work.

The most promising use for any address translation system is in virtualization. Recent work has shown that nested paging, direct hardware support for virtualized memory, comes with great performance penalty. While the use of very large pages (1GB on x86-64) would greatly reduce the frequency of TLB misses, they cannot typically be used because the hypervisor needs to maintain fine-grained control over guest physical address space permissions. Speculative address translation can allow the performance of large pages while maintaining fine-grained control.

Finally, we discuss future work related to maintaining the explicit reservation marks within the page table.

### 6.5.1 More aggressive reservation creation

Currently, FreeBSD only creates reservations for address space allocations that are expected to be of size greater than 2MB. If, for example, a process `mmap()`s a 1MB file, the

underlying memory is not be placed into a reservation. The operating system assumes that a file will not grow, once it is mapped into memory, so it could never fill a reservation and be promoted. Therefore, small pages are allocated to prevent memory fragmentation. However, with address speculation, reservations have performance benefits even when they are not promoted. A more aggressive reservation system could be developed to increase the fraction of small-page TLB misses that are able to be predicted. This would also require tweaks to `mmap()` to spread small files out in the virtual address space to ensure that each can receive its own reservation.

### **6.5.2 Very large page support for virtual machines**

To support hardware memory virtualization, x86-64 has introduced *nested paging*. Nested paging uses two sets of page tables, one maintained by the guest operating system and one maintained by the hypervisor. The guest page table translates guest virtual addresses into guest physical addresses. The hypervisor page table translates guest physical addresses into host physical addresses, as if the guest physical address space were yet another process on the host with its own virtual address space. This prevents the hypervisor from having to trap guest accesses to page tables. However, the cost of a TLB miss is greatly magnified because all guest physical memory accesses, both to data and the page table itself, must be translated through the hypervisor page table. This increases the number of total page table accesses required to translate an address from four to twenty-four [1]. This increase makes MMU caching vital. Bhargava et. al. [1] presented the AMD page walk cache (a unified page table cache) in the context of nested paging. They show that for the benchmarks investigated, the page walk cache is able to serve many of these requests. The scattering of pages within the guest physical address space means that the remaining memory accesses are to the hypervisor's page table.

Guest physical address space is generally allocated at VM start time, and is rarely swapped out. This would seem to make the hypervisor page table (translating guest physical to host physical addresses) an ideal candidate for the use of the very large page (1GB) support in x86-64. However, the hypervisor traps guest accesses to special regions of memory for I/O, including fixed location addresses within the “ISA hole”. This is done by marking regions of the hypervisor table as not present, causing a page fault when they are accessed.

While this precludes the use of a single very large *page*, this space can be stored in a 1GB reservation with both data and I/O pages mixed together. The SpecTLB can then provide speculative translations for access while the underlying address is validated in parallel. In the common case, the access will be to a data page and the speculative work can be committed from the reorder buffer. If the access turns out to have been to an I/O region, the speculative work is cancelled, and execution restarts. If very large page reservations were to be used, all of guest physical memory could be translated within a very small SpecTLB, eliminating the increased latency of nested paging.

Of course, large reservations lead to memory fragmentation. If a region of memory was reserved for a particular virtual machine, it could not be used by another. This would require guest physical memory to be static and an integral number of reservations in size. The impact of this tradeoff is left as future work.

### 6.5.3 Explicit marking

Explicit marking of pages that are part of a reservation can eliminate mispredictions that come from the inaccuracy of heuristically detecting reservations. With proper care built into the operating system, it can also ensure that no speculative accesses are made to I/O pages. When a page is inserted into the page table, if it is part of a large-page reservation,

the reservation bit is set. All speculative accesses made using that page are safe while the reservation exists. This has the drawback of being incompatible with the scheme set forth for virtualization above.

Eventually, if a reservation is not filled, it may be torn down. If the reservation hint bits remain set, the SpecTLB will predict the addresses for those pages that were added earlier. However, if any of the freed physical pages from the reservation are reused as I/O pages, the possibility exists that these pages may have speculative accesses made to them. Several possible solutions exist to this problem. Clearing all reservation bits related to a particular reservation when that reservation is torn down will prevent misspeculation, but it reduces the overall number of (possibly accurate) speculations that will be made. Additionally, reservation bits could be cleared when an I/O page is allocated. However, since a particular physical page can be mapped by multiple virtual pages (perhaps in different address spaces), a more complex map must be maintained to allow this to happen efficiently. Finally, even with explicit marking, the speculative request tagging discussed in Section 6.2.3 is still applicable. This would allow explicit marking and virtualization support to co-exist since the architecture would guarantee that speculative memory requests do not cause side-effects.

## 6.6 Conclusions

Large page support can dramatically reduce the performance overhead of virtual memory. For many applications, modern reservation based memory allocators do a good job of converting contiguously used regions of virtual address space into a manageable number of large pages. The TLB miss rates shown in this chapter are far lower than those for the small page simulations in chapter 4. However, what TLB misses remain are primarily from underfilled large page reservations. We have presented a device, the SpecTLB, that can

eliminate the latency from these misses and therefore eliminate the majority of the remaining performance penalty from a wide variety of applications running under FreeBSD.

The SpecTLB also enables a more radical implementation of reservation based memory allocation, one that reserves memory even when no promotion is possible. The most important application for this is virtualization, which can have particularly high overhead from virtual memory. Traditional hypervisors cannot readily use very large pages (1GB) to map guest physical memory due to fine-grained protection requirements. However, the SpecTLB can deliver the performance benefit of large pages while maintaining the protection granularity given by small pages.



Benchmark	ins/walk	Speculations			Mispredictions		DRAM Accesses Overlapped		TLB miss L2 hit rate
		total attempts	attempts/walk	successful/walk	per speculation	per walk	per walk	fraction of total	
astar	51888	1287	0.625	0.613	0.019	0.012	0.095	0.500	0.918
bwaves	8763	7918	0.963	0.962	0.001	0.001	0.143	0.897	0.910
bzip2	950	5391	0.293	0.293	0.001	0.000	0.039	0.271	0.293
cactus	16674	3017	0.848	0.848	0.000	0.000	0.279	0.830	0.305
calculix	72338	420	0.234	0.219	0.067	0.016	0.047	0.393	0.739
deal	32357	1134	0.472	0.464	0.016	0.007	0.017	0.355	0.860
gamess	40429	1450	0.590	0.590	0.000	0.000	0.013	0.440	0.926
gcc	8020	10517	0.852	0.841	0.012	0.010	0.289	0.760	0.152
Gems	51135	50	0.023	0.000	1.000	0.023	-	-	0.942
gobmk	8722	3652	0.279	0.277	0.006	0.002	0.099	0.430	0.491
gromacs	25142	3434	0.857	0.850	0.008	0.007	0.143	0.680	0.587
h264ref	18605	3879	0.785	0.784	0.001	0.001	0.086	0.610	0.562
hmmer	16145	4710	0.903	0.899	0.004	0.004	0.168	0.808	0.095
lbm	666336	2	0.015	0.015	0.000	0.000	0.015	0.027	0.371
leslie3d	17028	5931	0.932	0.932	0.000	0.000	0.215	0.868	0.068
libquantum	1275748	0	0.000	0.000	-	0.000	-	-	0.082
mcf	210	502927	0.992	0.992	0.000	0.000	0.143	0.982	0.214
milc	18490	4526	0.828	0.827	0.001	0.001	0.076	0.673	0.176
namd	44411	1436	0.518	0.512	0.011	0.006	0.051	0.448	0.740
omnetpp	5410	22872	0.754	0.750	0.006	0.005	0.102	0.688	0.316
povray	61836	528	0.375	0.375	0.000	0.000	0.016	0.407	0.915
sjeng	18433	2557	0.382	0.377	0.015	0.006	0.115	0.511	0.409
sphinx3	3740	31093	0.833	0.829	0.005	0.004	0.179	0.757	0.033
tonto	33002	1124	0.412	0.403	0.022	0.009	0.008	0.294	0.920
zeusmp	42484	4570	0.952	0.952	0.000	0.000	0.131	0.811	0.015

Table 6.1 : SpecTLB simulation results for SPEC CINT and CFP2006.

Benchmark	ins/walk	Speculations			Mispredictions		DRAM Accesses Overlapped		
		total attempts	attempts/walk	successful/walk	per speculation	per walk	per walk	fraction of total	TLB miss L2 hit rate
bt.C	109448	814	0.814	0.811	0.004	0.003	0.108	0.618	0.113
cg.C	33512	152	0.019	0.018	0.072	0.001	0.012	0.025	0.106
dc.B	43508	167	0.081	0.028	0.659	0.054	0.007	0.070	0.674
ep.C	23927	78	0.014	0.013	0.038	0.001	0.003	0.022	0.210
is.C	140525	860	0.721	0.718	0.005	0.003	0.102	0.603	0.917
lu.C	103736	795	0.742	0.740	0.003	0.002	0.103	0.556	0.438
sp.C	18346	5663	0.964	0.964	0.001	0.001	0.101	0.865	0.499
ua.C	117350	471	0.416	0.413	0.006	0.003	0.066	0.331	0.763
PostgreSQL (Fresh)	2450	29982	0.762	0.754	0.011	0.008	0.057	0.545	0.614
PostgreSQL (Cached)	12806	4215	0.567	0.561	0.010	0.006	0.129	0.543	0.360
Python 2.6	29200	3326	0.760	0.759	0.002	0.002	0.118	0.588	0.455
SPECjbb2005	5432	8853	0.418	0.406	0.029	0.012	0.135	0.354	0.401

Table 6.2 : SpecTLB simulation results for other benchmarks.

## CHAPTER 7

---

### Conclusions

---

Popular opinion maintains that since virtual memory is such an old problem that it is a solved problem. While much progress has been made on various techniques such as caching and large pages, the footprint and access pattern of popular workloads is making address translation more and more difficult. Large pages can cover a larger quantity of memory, but as available physical memory continues to increase, fractional TLB coverage shrinks. Page table structures based on hashing reduce the number of accesses required to the memory hierarchy, but show poor access locality. Therefore, they were useful when DRAM was not much slower than cache but are a poor choice for modern systems where caching performance is critical.

For applications with random access patterns, TLB hit rates are near zero for any page size. Traditionally, database hash joins have been an example of an application with a highly random access pattern, but other applications are starting to show similar behavior. Scientific computing is increasingly working with sparse and irregular data structures which have significant randomness in their access pattern [4]. TLB misses are, and will continue to be, a major source of performance overhead for important applications.

This thesis has examined three different techniques to reduce the cost of TLB misses with real-life workloads. These workloads exhibit significant locality in their virtual address use which maps to significantly more locality in physical address space usage than is

generally expected. This has great impact on which techniques are and are not effective at improving address translation performance.

Applications that cause a TLB miss on a particular virtual page will often cause a TLB miss on a nearby virtual page in the near future. Therefore, there is much reuse in the upper levels of the radix-tree page table. We examined many designs of *MMU cache* that store the upper levels of the page table. Such caches have been produced by the AMD and Intel in their processors, however their designs have not been previously compared in the literature. This has left significant optimization opportunity on the table. Unlike a processor data cache, an MMU cache has a very specific application, the caching of page table entries. This allows an MMU cache to be optimized for its exact task.

While AMD’s page walk cache (a unified page table cache) is conceptually simple, we showed that an different tag can be used to create a translation cache. This tag is both smaller and faster, since it requires fewer bits and allows page table levels to be skipped. The comparison of unified and split caches revealed the previously unpublished problem of page table level conflict. Intel’s split cache design avoids this problem at the expense of area. We have proposed a novel replacement policy, VI-LRU, that solves it by adapting to workload automatically. This replacement policy allows a unified MMU cache of  $n + 1$  entries to perform as well as a split cache holding a total of  $3 \times n$  entries, even under pathological workloads.

We showed that changes to the hardware have invalidated widely held beliefs about what is and is not a good page table format. Twenty years ago, memory hierarchies were flatter and the best metric of VM overhead was memory accesses. Since then, DRAM has become comparatively slower and caching of table entries has become more important. We show that the scattering of pages and tagging overhead in hashed page tables results in poor cache hit rates. While these formats require fewer memory hierarchy accesses overall,

the radix-tree allows more consecutive translations to fit into a single cache line. This greatly reduces the number of slow DRAM accesses required to translate virtual addresses as compared to an inverted page table. As hardware continues to evolve, these tradeoffs will continue to change.

With the availability of superpage support on x86, there has been a flurry of research on automatic support for page size selection. FreeBSD provides this with the reservation based memory allocator. We show that this software provides a unique opportunity to develop new hardware to predict address translations. Our device, the SpecTLB, interpolates a physical location for many TLB misses without any memory accesses. This allows speculative execution while a page walk occurs in the background to verify the predicted address. In our simulations, it is able to remove a per-benchmark average of 56% of MMU-related DRAM accesses from the critical path of execution.

The most obvious challenge for virtual memory is translating a virtual address into a physical address. The dependence on this translation is what causes the latency overhead of a TLB miss. However, it is the permissions and allocation that provides the biggest challenge to large page usage. An operating system cannot use a large page for a single region of virtual memory if parts of it are unallocated or need to have different permissions. Our device effectively decouples the address translation from the allocation and permission setting of large pages. The physical address can be interpolated immediately, allowing immediate execution, while allocation and permissions are verified in the background. The SpecTLB allows the fine-grained control of small pages while avoiding their latency penalty.

## 7.1 Future work

Nested paging presents a new set of challenges for MMU cache design. The AMD page walk cache utilizes a Nested TLB, however it may be possible to combine this device with a translation cache, or the primary TLB itself. Like moving from a split to a unified cache, this would increase flexibility to differing workloads. However, it may also admit an even more extreme version of level conflict since entries from entirely different page tables are being cached together. More work should be done to examine the behavior of different MMU cache designs under nested paging and to develop a proper replacement policy.

The new hardware proposed in this thesis similarly provides opportunity for more software research. Virtualization is a prime candidate for the very large page support now available in x86 microprocessors. While protection concerns previously made these impractical, the SpecTLB enables their use. Actually allocating and managing such pages is a research topic of its own. While there are likely thousands of 2MB pages available to a system, there are only a few 1GB pages in even the largest of machines. Allocating such a large slice of memory will require techniques to clean up address space fragmentation, such as those proposed by Romer *et al.*, ([16]).

Even in unvirtualized environments, the current behavior of the FreeBSD memory allocator does not exhaust the capability of the SpecTLB. The majority of TLB misses seen in our traces comes from small page misses, not large page misses. Of those, many physical addresses cannot be predicted, even with a very large SpecTLB. This means that reservations are not being created as often as they could be. Further research is necessary to determine if creating reservations more eagerly leads to more address prediction.

As address spaces continue to grow into the distant future, none of these techniques may be enough. While memory capacity is likely to continue to grow rapidly, I/O speed and

TLB size scale far slower. Selecting larger and larger page sizes will be impractical if these pages ever need to be read from or written to disk. This will require more radical solutions if future designers want to avoid the overhead of translating an address for nearly every memory access. Paged environments may not be practical at all, and architecture may have to return to using physical addressing. This would eliminate translation overhead entirely by simply removing translation. While impractical in current computing environments, a physical address space may be usable when all code runs under a trusted middleware environment, like Java.

The success of techniques developed here are due to their inspiration by actual software behavior. The traditional architectural design model of hardware dictating system support is flawed. Without examining the behavior of modern memory allocators, one would rightfully assume that there is no contiguity or alignment in physical address space usage. However, when a system is designed holistically, more complex behavior that can be exploited for performance becomes apparent.

## APPENDIX A

---

### MMU Coverage Proofs

---

The coverage provided by some of the MMU cache designs can be compared without making any assumptions about the access pattern. This section will show that the entries stored in the TPC with LRU replacement are always a superset of the entries stored in an equally sized unified cache (UPTC or UTC) with LRU replacement. This section will also show that the entries stored in a SPTC are always a superset of the entries stored in a TPC when all of the caches are equally sized and use LRU replacement.

Note, however, that coverage is not necessarily directly related to performance. Any entry that matches in a translation cache can be used immediately to begin the page walk from that point, skipping the previous steps of the page walk. In a page table cache, the traversal must occur in order, so the presence of a lower level entry in the cache does not allow the previous steps in the page walk to be skipped. So, a translation cache can outperform a page table cache, even in cases where the page table cache has larger coverage.

All of the MMU caches contain physical page numbers of page table pages that could be accessed during a page walk. The page walk is the traversal of a radix tree with the physical page number of the root stored in the CR3 register, as shown in Figure 2.2. CR3 points to an L4 page table page containing L4 entries. Each L4 entry, in turn, is either valid and points to an L3 page table page or is invalid (*i.e.*, NULL) and indicates that there are no valid virtual addresses with that index. Similarly, each valid L3 entry points to an L2



page table page. Then, each valid L2 entry points to an L1 page table page. Finally, each valid L1 entry contains the physical page number of the data page. The offset of the virtual address is appended to this physical page number to create the address of the data itself. As shown in Figure 2.2, L1 entries are not cached by the MMU caches, as they are cached by the TLB. Therefore, L1 entries will be ignored for the remainder of this section.

The following definitions are used by the subsequent proofs:

**Definitions:** An *internal entry* is an L4 or L3 entry which can be cached in any of the MMU caches. A *leaf entry* is an L2 entry which is the lowest level of the tree that is cached by any MMU cache. An *entry* is any internal entry or leaf entry. Finally, a *path* is a sequence of L4, L3, and L2 entries that are a part of a single page walk.

## A.1 Comparison of TPC and Unified Coverage

On a TLB miss, the MMU must walk the page table tree from the root all the way down to the L1 entries. Each slot in the TPC stores the entire path of such a walk, whereas each slot in a unified cache (either a UPTC or UTC) stores a single entry in such a walk. While these strategies are quite different, this section will prove that given a TPC and unified cache of equal size that both use a least-recently used (LRU) replacement policy, the entries stored in the TPC are always a superset of the entries stored in the unified cache. The following lemmas will be used in support of this proof:

**Lemma A.1.1** *For any sequence of page walks, any leaf entry cached by any  $n$ -entry cache using LRU replacement would also be cached by an  $n$ -path TPC using LRU replacement.*

**Proof** The TPC caches entire paths using an LRU replacement policy, so it will always contain the  $n$  most recently walked unique paths. As each path includes a leaf entry, the TPC will therefore always contain the  $n$  most recently accessed unique leaf entries. By

definition, any  $n$ -entry cache using an LRU replacement policy cannot cache any entry that was accessed less recently than the  $n$  most recently accessed leaf entries. Therefore, no  $n$ -entry LRU cache can ever cache a leaf entry that would not also be cached by an  $n$ -path TPC ■

**Lemma A.1.2** *In a unified cache using LRU replacement, no internal entry can be present in the cache without at least one of its child leaf entries also being present in the cache.*

**Proof** Entries are only accessed as a part of a complete page walk. Therefore, leaf entries of a path will always be accessed after internal entries in that path. Therefore, the leaf entry of any path is always more recently accessed than the internal entries of that path. So, due to the LRU replacement policy, when entries are replaced in any LRU unified cache, the leaf entry of a path will always be replaced after its parent internal entries. Therefore, at any time, no internal entry can be in an LRU unified cache without at least one of its child leaf entries also being in the LRU unified cache. ■

**Theorem A.1.3** *For any sequence of page walks, any entry cached by an  $n$ -entry unified cache (UPTC or UTC) using LRU replacement would also be cached by an  $n$ -path TPC using LRU replacement.*

**Proof** By Lemma A.1.1, an  $n$ -path LRU TPC always caches a superset of the leaf entries cached by an  $n$ -entry LRU unified cache. Furthermore, because the TPC caches complete paths, it always caches every parent internal entry of each leaf entry in the TPC. By Lemma A.1.2, a unified cache can never cache an internal entry that is not the parent of a leaf entry within the UPTC. Since those leaf entries are a subset of the leaf entries in the TPC, the internal entries in the unified cache must also be a subset of the internal entries in the TPC. Therefore, the entries cached within an  $n$ -entry LRU unified cache are always a subset of the entries cached within an  $n$ -path LRU TPC. ■

## A.2 Comparison of TPC and SPTC Coverage

While each slot of the TPC caches the entire path of each page walk, the SPTC caches the entries of each level of the walk in a separate cache. If each of the SPTC caches have the same number of slots as the TPC and all of the caches use an LRU replacement policy, this section will prove that the entries stored in an SPTC with  $n$  entries for each level will always be a superset of the entries stored in an  $n$ -path TPC. The following lemmas will be used in support of this proof:

**Lemma A.2.1** *For any sequence of page walks, the leaf entries cached in an  $n$ -path LRU TPC are always identical to the leaf entries cached in an LRU SPTC with  $n$  entries in its L2 entry cache.*

**Proof** As stated in Lemma A.1.1, an  $n$ -path LRU TPC always contains the  $n$  most recently accessed unique leaf entries. Similarly, the SPTC caches leaf entries in its L2 entry cache using an LRU replacement policy. Since every page walk using an SPTC must access a leaf entry, the L2 entry cache of the SPTC will always contain the  $n$  most recently accessed unique leaf entries. Therefore, the leaf entries cached by each structure will always be identical. ■

**Lemma A.2.2** *For any sequence of page walks, any internal entry cached by an  $n$ -path LRU TPC would also be cached by an LRU SPTC with  $n$  entries for each level.*

**Proof** When using an SPTC, every page walk will access all the entries in the path. As the SPTC has  $n$  entries for each level and uses an LRU replacement policy, this means that the SPTC will always contain the  $n$  most recently accessed unique internal entries at each level. As the TPC stores  $n$  unique paths, at most  $n$  unique internal entries can be cached at each level. The LRU replacement policy of the TPC guarantees that none of these internal

entries can ever be less recently accessed than the  $n$  most recently accessed internal entries at that level. Therefore, the TPC can never cache an internal entry that would not also be cached by the SPTC. ■

**Theorem A.2.3** *For any sequence of page walks, any entry cached by an  $n$ -path LRU TPC would also be cached by an LRU SPTC with  $n$  entries for each level.*

**Proof** By Lemma A.2.1, the leaf entries cached by each structure will always be the same. By Lemma A.2.2, the TPC can never cache an internal entry that would not also be cached by the SPTC. Therefore, an  $n$ -path LRU TPC can never cache any entry that would not also be cached by an LRU SPTC with  $n$  entries for each level. ■

Note that if the L4 and/or L3 entry caches of the SPTC are smaller than the L2 entry cache, then this theorem does not hold because Lemma A.2.2 requires all of the SPTC's caches to be the same size. When they are not, the SPTC may hold a superset, subset, or the same set of internal entries that the TPC holds, depending on the access pattern.

---

## Bibliography

---

- [1] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, “Accelerating two-dimensional page walks for virtualized systems,” in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 26–35, ACM, 2008.
- [2] C. McCurdy, A. L. Cox, and J. Vetter, “Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors,” in *ISPASS ’08: Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, (Washington, DC, USA), pp. 95–104, IEEE Computer Society, 2008.
- [3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, “DBMSs on a Modern Processor: Where Does Time Go?,” in *VLDB ’99: Proceedings of the 25th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 266–277, Morgan Kaufmann Publishers Inc., 1999.
- [4] R. C. Murphy and P. M. Kogge, “On the memory access patterns of supercomputer applications: Benchmark selection and its implications,” *IEEE Trans. Comput.*, vol. 56, no. 7, pp. 937–945, 2007.
- [5] J. Navarro, S. Iyer, P. Druschel, and A. Cox, “Practical, transparent operating system support for superpages,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 89–104, 2002.
- [6] M. Talluri and M. D. Hill, “Surpassing the tlb performance of superpages with less operating system support,” in *Proceedings of the Sixth International Conference on*

*Architectural Support for Programming Languages and Operating Systems*, 1994.

- [7] M. Talluri, M. D. Hill, and Y. A. Khalidi, “A new page table for 64-bit address spaces,” in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 184–200, ACM, 1995.
- [8] Advanced Micro Devices, *AMD x86-64 Architecture Programmer's Manual, Volume 2*, 2002.
- [9] K. Bala, M. F. Kaashoek, and W. E. Weihl, “Software prefetching and caching for translation lookaside buffers,” in *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, (Berkeley, CA, USA), p. 18, USENIX Association, 1994.
- [10] M. Wu and W. Zwaenepoel, “Improving tlb miss handling with page table pointer caches,” Tech. Rep. TR97-296, Rice University, 1996.
- [11] B. L. Jacob and T. N. Mudge, “A look at several memory management units, TLB-refill mechanisms, and page table organizations,” in *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 295–306, ACM, 1998.
- [12] J. Liedtke, “Address space sparsity and fine granularity,” in *EW 6: Proceedings of the 6th workshop on ACM SIGOPS European workshop*, (New York, NY, USA), pp. 78–81, ACM, 1994.
- [13] N. Ganapathy and C. Schimmel, “General purpose operating system support for multiple page sizes,” in *In Proceedings of the USENIX Conference. USENIX*, pp. 91–104, 1998.

- [14] J. Mauro and R. McDougall, *Solaris Internals*. Sun Microsystems Press, 2000.
- [15] C. Mather, I. Subramanian, I. Subramanian, C. Mather, K. Peterson, K. Peterson, B. Raghunath, and B. Raghunath, “Implementation of Multiple Pagesize Support in HP-UX,” 1998.
- [16] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, “Reducing TLB and memory overhead using online superpage promotion,” in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, (New York, NY, USA), pp. 176–187, ACM, 1995.
- [17] A. Saulsbury, F. Dahlgren, and P. Stenström, “Recency-based tlb preloading,” in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 117–127, ACM, 2000.
- [18] G. B. Kandiraju and A. Sivasubramaniam, “Going the distance for tlb prefetching: an application-driven study,” in *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, (Washington, DC, USA), pp. 195–206, IEEE Computer Society, 2002.
- [19] A. Bhattacharjee and M. Martonosi, “Inter-core cooperative tlb for chip multiprocessors,” in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 359–370, ACM, 2010.
- [20] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide Part 1*, 2009.
- [21] N. Young, “On-line caching as cache size varies,” in *SODA '91: Proceedings of the*

- second annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 241–250, Society for Industrial and Applied Mathematics, 1991.
- [22] P. Shyamkumar and N. P. Jouppi, “CACTI 3.0,” Tech. Rep. WRL 2001/2, Compaq Western Research Labs, August, 2001.
- [23] R. Bedichek, “SimNow: Fast platform simulation purely in software,” in *Hot Chips 16*, 2004.
- [24] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.
- [25] Standard Performance Evaluation Corporation, “The SPEC JBB2005 Benchmark,” 2005.
- [26] *The ASCI sweep3d Benchmark Code*.
- [27] J. J. Dongarra and P. Luszczek, “Introduction to the HPCChallenge Benchmark Suite,” Tech. Rep. 05-544, University of Tennessee - Knoxville, <http://icl.cs.utk.edu/hpcc/>, 2005.
- [28] J. Edler and M. D. Hill, “Dinero IV Trace-Driven Uniprocessor Cache Simulator,” 1998.
- [29] H. Garcia-Molina, J. Ullman, and J. Widom, *Database System Implementation*. Prentice Hall, 2000.
- [30] D. E. Knuth, *The Art of Computer Programming 3. Sorting and Searching: The Classic Work Newly Updated and Revised*. Addison-Wesley Longman, Amsterdam, 2. a. ed., 1998.



- [31] Intel Corporation, *Intel Itanium Architecture Software Developer's Manual - Volume 2: System Architecture, Revision 2.2*, 2006.
- [32] Sun Microsystems, *UltraSPARC III Cu User's Manual*, 2004.
- [33] International Business Machines, Inc., *The PowerPC Architecture*, 2nd ed., 1994.
- [34] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrisnan, and S. K. Weeratunga, "The nas parallel benchmarks," tech. rep., The International Journal of Supercomputer Applications, 1991.
- [35] M. Kirkwood. <http://benchw.sourceforge.net/>.