The Dissertation Committee for Tianhao Zheng
certifies that this is the approved version of the following dissertation:

Efficient Fine-grained Virtual Memory

Committee:

Mattan Erez, Supervisor

Vijay Janapa Reddi

Calvin Lin

Simon Peter

Mohit Tiwari

Efficient Fine-grained Virtual Memory

by

Tianhao Zheng

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2018

To my loving parents, Zhenhong Zheng and Yanru Tang.

Acknowledgments

My Ph.D. journey would not have been possible without tremendous supports and guidance from my advisor, Mattan Erez. He was smart, knowledgable, hard-working and the most important, open-minded and willing to help all the way through my PhD career. Instead of assigning a direction to me, he inspired me to grow my own research interest from the beginning of the Ph.D. program. I shall always remember the days I started with little experience, making numerous mistakes on various ideas and experiments. He was always patient and supportive, happy to spend time on my questions no matter how trivial they are.

I thank Vijay Janapa Reddi, Mohit Tiwari, Calvin Lin, Simon Peter for serving on my dissertation committee and providing me valuable comments on this dissertation.

I would like to thank David Nellans, Arslan Zulfiqar, Mark Stephenson and Stephen W. Keckler at Nvidia Research for the opportunity to collaborate with them. I learned a lot about GPU memory systems while I was interning with them.

I am glad to get to know many other graduate students during my PhD. Specifically, I would like to thank Jaeyoung Park, Dong Wan Kim, and Haishan Zhu for the collaborations with them. Their knowledge and experience expended my expertise and helped me to develop many research ideas.

I would also like to acknowledge the old members of LPH group: Min Kyu Jeong, Mike Sullivan, Evgeni Krimer, Minsoo Rhu, Ikhwan Lee, Jungrae Kim for selflessly sharing their experience and knowledge with me.

I also thank all the other members of our research group: Song Zhang, Jinsuk Chung, Seong-Lyong Gong, Kyushick Lee, Esha Choukse, Derong Liu, Nick Kelly, Benjamin Cho, Sangkug Lym, Yongkee Kwon, Chun-Kai Chang, Majid Jalili, and Wenqi Yin. And I am grateful for the friendship with many others: Yuhao Zhu, Jingwen Leng, Yazhou Zu, Wenzhi Cui, Matt Halpern, Zhuoran Zhao, Shuang Song, Qinzhe Wu, Hangchen Yu, Pei Yan. It has been much fun having coffee and pizza together and spend days and nights on projects.

Finally, I cannot express enough my gratitude to my parents, Zhenhong Zheng and Yanru Tang for their unconditional love. I would like to appreciate my girlfriend, Yawen He for the adventures we took together.

Tianhao Zheng
April 2018, Austin, TX

Efficient Fine-grained Virtual Memory

Publication No. ___________________

Tianhao Zheng, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Mattan Erez

Virtual memory in modern computer systems provides a single abstraction of the memory hierarchy. By hiding fragmentation and overlays of physical memory, virtual memory frees applications from managing physical memory and improves programmability. However, virtual memory often introduces noticeable overhead. State-of-the-art systems use a paged virtual memory that maps virtual addresses to physical addresses in page granularity (typically 4 KiB).This mapping is stored as a page table. Before accessing physically addressed memory, the page table is accessed to translate virtual addresses to physical addresses. Research shows that the overhead of accessing the page table can even exceed the execution time for some important applications. In addition, this fine-grained mapping changes the access patterns between virtual and physical address spaces, introducing difficulties to many architecture techniques, such as caches and prefecthers.

In this dissertation, I propose architecture mechanisms to reduce the overhead of accessing and managing fine-grained virtual memory without compromising existing benefits. There are three main contributions in this dissertation.

First, I investigate the impact of address translation on cache. I examine the restriction of virtually indexed, physically tagged (VIPT) caches with fine-grained paging and conclude that this restriction may lead to sub-optimal cache designs. I introduce a novel cache strategy, speculatively indexed, physically tagged (SIPT) to enable flexible cache indexing under fine-grained page mapping. SIPT speculates on the value of a few more index bits (1 - 3 in our experiments) to access the cache speculatively before translation, and then verify that the physical tag matches after translation. Utilizing the fact that a simple relation generally exists between virtual and physical addresses, because memory allocators often exhibit contiguity, I also propose low-cost mechanisms to predict and correct potential mis-speculations.

Next, I focus on reducing the overhead of address translation for fine-grained virtual memory. I propose a novel architecture mechanism, Embedded Page Translation Information (EMPTI), to provide general fine-grained page translation information on top of coarse-grained virtual memory.

EMPTI does so by speculating that a virtual address is mapped to a pre-determined physical location and then verifying the translation with a very-low-cost access to metadata embedded with data. Coarse-grained virtual memory mechanisms (e.g., segmentation) are used to suggest the pre-

determined physical location for each virtual page. Overall, EMPTI achieves the benefits of low overhead translation while keeping the flexibility and programmability of fine-grained paging.

Finally, I improve the efficiency of metadata caching based on the fact that memory mapping contiguity generally exists beyond a page boundary. In state-of-the-art architectures, caches treat PTEs (page table entries) as regular data. Although this is simple and straightforward, it fails to maximize the storage efficiency of metadata. Each page in the contiguously mapped region costs a full 8-byte PTE. However, the delta between virtual addresses and physical addresses remain the same and most metadata are identical. I propose a novel microarchitectural mechanism that expands the effective PTE storage in the last-level-cache (LLC) and reduces the number of page-walk accesses that miss the LLC.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Virtual memory has been introduced to computer systems for efficiency, programmability, and security. It creates a contiguous address space for each process, such that programmers are not required to deal with fragmentation or overlays of physical memory explicitly. In addition, virtual memory enables fine granularity memory mapping. In state-of-the-art systems, this mapping is usually achieved with 4 KiB pages. By mapping virtual memory to physical memory with such fine-grained pages, non-contiguous physical memory can be efficiently utilized to satisfy contiguous virtual memory allocations. Virtual memory also provides various metadata, including access permissions, page attributes, and access information, which are necessary for correct and efficient system operation. In fact many additional uses of metadata such as tracking access counts [59] and monitoring memory accesses [85] have been proposed.

1.1 The Cost of Fine-Grained Paged Virtual Memory

Everything comes at a cost. State-of-the-art virtual memory cannot provide all features mentioned above while keeping the overhead low. Three major issues with current virtual memory systems are summarized below:

(1) The fine-grained virtual to physical mapping may change access patterns between virtual and physical address spaces. This indirection brings difficulties to many architecture techniques. For example, virtually indexed, physically tagged (VIPT) caches can only be indexed with partial addresses within a page boundary. For physical address based prefetching mechanisms, crossing page boundaries can be harmful [23]. These either directly hurt performance, or constrain trade-off options, leading to sub-optimal designs.

(2) Main memory and most caches are physically addressed. Before accessing them, address translations must be performed. In standard 4 KiB paged virtual memory, the memory mapping is stored in page tables with each entry containing the translation information and other metadata for a page. To reduce page table storage, radix tree structures are employed in most architectures, including x86 [41] and ARM [6]. Each translation is obtained by a page walk with multiple sequential memory look-ups. Mechanisms such as translation lookaside buffers (TLBs) and memory-management unit (MMU) caches are used to accelerate this look-up process. But these on-chip structures cannot be scaled up efficiently. For page walk intensive workloads, up to 83.1% of execution time may be spend on page walks in a processor with a 2-level TLB and MMU caches [9].

(3) Metadata caching is critical. While dedicated buffering structures, such as TLB and MMU caches, are used for better performance, the total capacity is constrained by limited on-chip storage. Caching metadata in the regular data cache hierarchy also plays an important role in reducing the la-

tency of address translation [64, 72]. Current caching schemes for metadata that simply treat metadata as regular data fail to maximize storage efficiency. The redundancy of page-grained metadata has been investigated by Basu et al. [9] who find that the per-page permissions are identical for vast majority of pages. Prior research [89, 70] also suggests that the contiguity of address mapping generally exists beyond a page boundary. Therefore, the mapping of many pages in the same contiguously-mapped region can be represented with a single delta between virtual addresses and physical addresses and a single set of permissions.

1.2 The Real Implications of Fine Granularity

Enormous effort has been spent on reducing the cost of fine-grained virtual memory, mainly in two directions. The first is to strictly keep fine-grained virtual memory and focus on improving microarchitectural support, such as introducing larger and more efficient TLBs and MMU caches. The second direction completely gives up on fine granularity and employs a coarser granularity, exchanging flexibility, compatibility, programability, and fine-grained metadata for lower overhead. Instead of going to either extreme, I identify better tradeoffs between fine and coarse granularity memory management. As an example, previous work [70] and my experiments (Figure 6.2) show that memory mapping is frequently contiguous across page boundaries, even with the standard 4KiB paged virtual memory. Maintaining fine-grained mapping within a contiguously-mapped region is not necessary, yet it wastes storage space and incurs runtime overhead. At the same time, coarser memory mapping granularity can be carefully used within the contiguously-mapped region without hurting the flexibility of fine-grained memory mapping. In this dissertation, I explore a pay-as-you-go approach for supporting fine-grained pages. Compatibility-wise, I do not compromise the ability to use fine-grained paging. Programmers are always free from porting source code. Performance-wise, I propose various schemes to remove the unnecessary overhead, utilize the "free" contiguity mentioned above, and make sure the cost of fine-grained virtual memory is paid only when necessary.

1.3 Thesis Statement

Current fine-grained paged virtual memory provides appealing features with significant overhead, e.g., long translation latency and changed access patterns. Future virtual memory systems can reduce this overhead and approach the overhead of coarse-grained paging without compromising programmability, efficiency, and metadata.

1.4 Contributions

The goal of my dissertation is to enable efficient fine-grained paged virtual memory mechanisms that resolve the issues mentioned above, without compromising existing benefits. To achieve this goal, I develop user-transparent, low-overhead architecture mechanisms on top of existing, standard computer systems that improve system efficiency while maintaining backward compatibility. In the first part of this dissertation I analyze issues in existing state-of-the-art systems, and evaluate their impact. In the second part of the dissertation, I propose various novel architecture mechanisms to address these issues, quantitatively evaluate the effectiveness of proposed mechanisms, and compare with prior work. To summarize the main contributions of my dissertation:

- I examine the restriction of virtually indexed, physically tagged (VIPT) caches with fine-grained paging and conclude that this restriction may lead to suboptimal cache designs. I introduce a novel cache strategy, speculatively indexed, physically tagged (SIPT) to enable flexible cache indexing under fine-grained page mappings. SIPT speculates on the value of a few more index bits (1 - 3 in our experiments) to access the cache speculatively before translation and then verify the physical tag match after translation. Utilizing the fact that a simple relation generally exists between virtual and physical addresses, because memory allocators often exhibit contiguity, I also propose low-cost mechanisms to predict and correct potential mis-speculations.

- I propose a novel architecture mechanism, Embedded Page Translation Information (EMPTI), to provide general fine-grained page translation information on top of coarse-grained virtual memory. EMPTI does so by speculating that a virtual address is mapped to a pre-determined physical location and then verifying the translation with a very-low-cost access to metadata that is embedded with the data. Coarse-grained virtual memory mechanisms (e.g. segmentation) are used to suggest the pre-determined physical location for each virtual page. Overall, EMPTI achieves the benefits of low overhead translation while keeping the flexibility and programmability of fine-grained paging.

- I revisited prior research on address mapping contiguity and redundancy of per-page permissions, and propose Delta Caching to achieve more efficient, yet flexible metadata caching. Delta caching provides up to 4× higher storage density than when 4KiB page table entries are stored as data, and does not compromise mapping flexibility and fine-grained metadata. By converting DRAM accesses incurred during page walks into LLC hits, delta caching removes a substantial fraction of page walk overhead, when combined with existing techniques, Delta Caching outperforms THP.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 provides background on virtual memory and clarifies terminologies; Chapter 3 discusses challenges this dissertation tries to address; Chapter 4 revisits design constrains in virtually indexed, physically tagged (VIPT) caches. I propose speculatively indexed, physically tagged (SIPT) caches to remove these constrains. I evaluate the benefit in terms of both performance and energy. Chapter 5 focus on reducing the cost of virtual to physical translation. I present and evaluate Embedded Page Translation Information (EMPTI). With EMPTI, low translation overhead and fine-grained paging can be achieved at the same time. Chapter 6 address the inefficiency of current metadata storage in the data cache hierarchy by introducing Delta Caching. Chapter 7 concludes this dissertation.

Chapter 2

Background and Terminology

2.1 Fine-Grained Virtual Memory

The standard 4 KiB paged virtual memory has evolved into the dominant memory management scheme. Its strength is that virtual pages are a single abstraction for all memory-management needs and their fine-grained nature and flexible mapping enable numerous optimizations and system functions. Page-based virtual memory works by having applications only use virtual addresses, which are then mapped to physical addresses via an OS-maintained page table. Because of increasing memory capacities, current systems utilize a multi-level hierarchical page table structure. As an example, current 64-bit x86 processors from Intel [41] and AMD [1] use a 4-level page table.

Before accessing data, a memory operation must translate the virtual address into a physical one, requiring a page walk with up to 4 memory accesses for address translation, on top of the memory access for the data (Figure 2.1). For each level, the corresponding entry can be located by adding an offset from part of the virtual address to the base address from a pre-defined register (CR3 in x86) or the result of last level. By repeating this look-up 4 times, an L1 Page Table Entry (PTE, Figure 2.2) from the hierarchical page table is

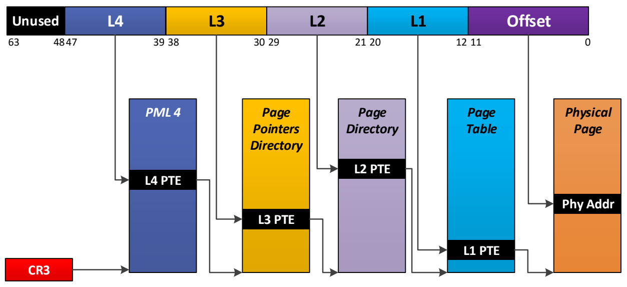


Figure 2.1: 4-level page walk in x86 architecture

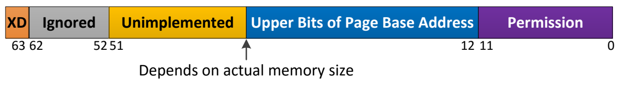

XD
Ignored
Unimplemented
Upper Bits of Page Base Address
Permission
63 62
52 51
12 11
0
Depends on actual memory size

Figure 2.2: A x86 page table entry

loaded to the MMU. The MMU then decodes the PTE, and uses the upper bits of the page base address or physical frame number (PFN) to locate the physical base address of the page. The metadata in the permission bits are also checked. With the PFN and offset bits within the virtual address, the actual data can then be located and accessed in physical memory.

2.1.1 Translation Lookaside Buffers

To overcome the latency of indirection, translation lookaside buffers (TLBs) are used [29]. By caching address mapping information in a dedicated hardware buffer, TLBs eliminate the overhead of fetching paging information from memory in most cases. However, some applications have poor spatial locality and exhibit frequent TLB misses, which significantly impairs application performance as memory is repeatedly accessed to perform translations [57, 9].

Various techniques have been proposed to improve the efficiency of

TLBs. TLB clustering [69] enlarge the reach of a TLB by storing mapping of contiguous pages as one TLB entry. Prefetching is another technique that has been proposed for improving TLB hit rate and shows benefits for some applications [44, 55, 76]. However, the effectiveness of these techniques is limited by their ability to identify access patterns, which proves challenging for applications that make irregular memory accesses.

An early alternative to TLBs is the use of *inverted page tables* [16]. Instead of caching translations on chip, inverted page tables act as a large in-memory cache of translations. Part of the virtual address is used directly, or after hashing, as a physical address into the inverted page table, and on a hit, the inverted page table returns a tag to verify whether the translation information is valid for the virtual address that queried it. Inverted page tables decrease, but do not eliminate, the number of memory accesses required to fill a TLB entry.

2.1.2 Memory-Management Unit (MMU) Caches

Another approach is to utilize the hierarchical page structure to increase coverage by caching page directory entries (L2 PTEs, L3 PTEs, and L4 PTEs in Figure 2.1) in a dedicated MMU cache [8, 12, 41, 11]. This approach increases translation caching coverage effectively for some applications, but still suffers from memory accesses for page table levels that are closer to the leaves of the page table radix tree.

2.2 Coarse-Grained Virtual Memory

The standard 4 KiB paged virtual memory provides appealing features such as flexible address mapping and fine-grained metadata. However, address translation in such a fine granularity can still be very costly even with state-of-the-art processors with a 2-level TLB and MMU caches [9]. In contrast, another solution is to coarsen the granularity of address mapping. Although various schemes have been previously proposed or implemented, in general, there are three approaches to employ coarser granularity. One big issue with all these schemes is compatibility with many existing system features and optimizations.

2.2.1 Direct Segments

Direct segments [18, 20, 9] use range-based translation, in which large contiguous virtual regions are mapped to contiguous physical ones. Regions and segments make translation simple and practically eliminate all translation overhead. However, the absence of fine granularity protection and mapping does place limitation on system management and application characteristics. This can significantly limit their applicability. Developers need to find memory allocations that can be safely managed by direct segments. Source code modification is also required to opt-in or opt-out of direct segments explicitly. And, there is significant porting effort of libraries. This deficiency can be addressed by increasing the number of regions and thus more closely approaching the flexibility of fine-grained pages only when necessary. However, when the number of regions grow, the same issues as TLB coverage and TLB misses arise. Segments have been popular in the past for specialized high-performance and embedded systems, but have lost popularity to ubiquitous standardized paging schemes or have been combined with paging [73, 74].

2.2.2 Coarse-Grained Pages

Superpages [82, 81, 63] and hugepages [14, 53, 5] enlarge the standard translation granularity to improve the effectiveness of the TLB and reduce translation overhead. By enlarging the granularity of address mapping and management, the coverage of the TLB can be increased. However, larger page sizes increase the working set size [83], rely on large contiguous memory regions, and fail to provide fine granularity protection. Holes between small non-continuous regions can lead to wasted memory. A single 4KiB dirty page can cause the write back of the whole mostly clean large page in current hardware implementations [5, 14]. Even for applications with suitable memory behavior, it is challenging to use coarse-grained pages because the optimal page size depends on the application, system, and dynamic characteristics of the inputs. Note that academic research on superpages addresses some of these limitations, but not all [82, 81]. For example, in NUMA systems, large pages may lead to performance loss due to load imbalance and poor locality, which might entirely offset the benefits from fewer page walks [27]. In addition, forming coarser pages requires management in the OS and adds overhead that can be significant. This is apparent with the THP mechanism of

Linux for which I measure significant management costs with some benchmarks (e.g., 24% for IS.C and 10% for mcf), and the active effort to improve OS support [52].

2.2.3 Redundant Memory Mappings

Redundant Memory Mappings (RMM) [46] maintain both region-based translation information and fine-grained pages for the same addresses to improve performance while maintaining compatibility when fine-grained pages are a necessity. The TLB reach is increased with a range TLB and the regular TLB can take care of any non-contiguous spaces. However, the fine granularity metadata is not available in a range TLB and extra OS-management effort is required. For example, the OS may be required to allocate consecutive physical pages to consecutive virtual pages eagerly at allocation time and set accessed and dirty bits at allocation time, instead of relying on hardware to maintain this information dynamically, increasing overhead and potentially significantly coarsening the granularity of writing data out of memory. The OS must also maintain and apply policies for breaking regions and coalescing pages.

2.3 L1 Cache Indexing

Due to its critical impact on performance, much effort has been invested to improve L1 performance. The L1 presents challenging tradeoffs between hit rate and access latency. Access latency includes the virtual memory address translation latency (TLB lookup), tag array access and matching, and the data access itself. In order to push latency down, all three components should ideally overlap. Tag and data accesses are overlapped by accessing all ways simultaneously and delivering only tag-matching data (with a switch or multiplexer). Overlapping those two accesses with address translation is more challenging because an access can not start before the address is known. The simplest cache design indeed performs translation before L1 access begins. This design is called a *physically-indexed physically-tagged* (PIPT) cache because virtual addresses (VAs) are not used at all in the L1. While simple, the translation overhead is not hidden and access latency is often considered too high. Current designs solve the latency problem and enable access and translation overlap in one of two ways.

2.3.1 Virtually Indexed, Virtually Tagged Caches

The first solution is to translate virtual addresses (VAs) after the L1 is filled, thus accessing the cache purely with VAs. This design is known as *virtually-indexed virtually-tagged* (VIVT) and eliminates the translation latency for accessing the cache. However, relying purely on VAs for most memory accesses (as most are L1 hits) presents significant complications for cache management and coherence because software maps multiple VAs to the same physical address (*synonyms*) and may also map the same virtual address to multiple physical addresses across different address spaces (*homonyms*). Prior work has developed solutions, but the design is more complicated than the VIPT alternative, described below [48, 30].

2.3.2 Virtually Indexed, Physically Tagged Caches

The second solution relies only on the offset bits of the VA for computing L1 array locations; the offset bits are not translated and can hence be used at the same time translation proceeds. Before data is delivered, the tag is compared to the fully translated physical address (PA). This design is called *virtually-indexed physically-tagged* (VIPT). Virtually indexed, physically tagged (VIPT) caches are more appealing and popular among all three variants (PIPT, VIVT and VIPT) because they combine the strong correctness and simple coherence guarantees of PIPT caches with practically zero-latency translation. All addresses have the full physical address available through the tags for correctness and coherence. At the same time, the latency of translation can be fully hidden by accessing the cache arrays in parallel with only the page offset bits that are never modified by address translation. However, the important tradeoff made is the constraints on cache parameters. Specifically, each set is limited in capacity to a single virtual memory page. Therefore, the cache capacity is coupled with its associativity: capacity = #ways $\times$ 4KiB, assuming common 4KiB page granularity. For example, many current processors have 32KiB 8-way set-associative caches [10, 15, 19]. While high associativity reduces the conflict miss rate it also adds latency, which is potentially a suboptimal design point when compared to a larger lower-associativity cache. Furthermore, L1 cache access energy is also coupled with associativity because all ways are typically accessed in parallel to reduce latency.

Chapter 3

Problem Statement and Motivation

3.1 VIPT Cache Design Space with Fine-Grained Page Mapping

With a conventional virtually indexed, physically tagged (VIPT) data L1 cache, the cache associativity is inflated in order to provide sufficient capacity while meeting the VIPT indexing constraint. For example, Intel Haswell processors, have a 32KiB 8-way set-associative L1 data cache with 4-cycle access latency. This constraint in VIPT caches design may lead to suboptimal configurations.

I explore the capacity and associativity design space with Cacti 6.5 [62], and use the Haswell L1 cache configuration as the baseline. I simulate L1 caches with the configurations and parameters summarized in Table 3.1 and present the latency of the different configurations relative to the baseline in Figure 3.1. For each capacity and associativity, I sweep the number of read ports and the number of banks and show the range and mean of relative latencies for each configuration. The baseline for each bank/port configuration is always 32KiB with 8 ways.

While both capacity and associativity affect latency, associativity has

Technology	32 nm
Cache line size	64 Bytes
Capacity	16 KiB, 32 KiB, 64 KiB, 128 KiB
Associativity	2-way, 4-way, 8-way, 16-way, 32-way
Access mode	Parallel data and tag access
Ports	1 or 2 for read, 1 for write
Banks	1, 2 or 4 banks

Table 3.1: L1 cache configurations.

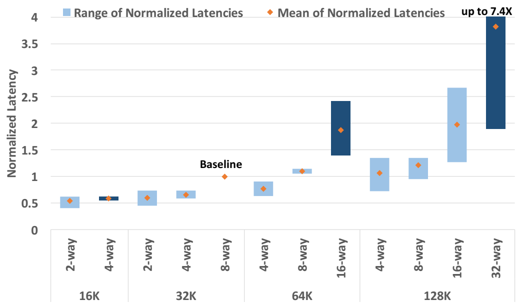


Figure 3.1: L1 cache latency relative to 32KiB 8-way baseline.

the greater impact. This is especially the case when increasing associativity from 4 to 8 ways. For example, we can reduce the latency of a 32KiB cache to 2 cycles by reducing its associativity to 2 ways. However, this configuration is not possible with VIPT and 4KiB pages because it needs 13 index bits, one more than the 12 offset bits of the page. Configurations that are not feasible for VIPT are shaded light blue while those that are feasible are shaded dark blue. Unfortunately, perhaps the most desirable configurations are currently infeasible.

The next question is what impact these tradeoffs have on application performance. I run applications from SPEC CPU 2006 [35] suite and SPEC CPU INT 2017 [80] with reference inputs. In addition to the large-memory footprint applications of SPEC INT 2017 (>8GiB for many), I also evaluate large-memory big-data applications: graph500 (graph processing) [32] and DBx1000 with the ycsb workload (database) [87], each configured to use more than 4GiB of memory. For each application I collect 500 million instructions at a SimPoint [79]. While different applications exhibit different characteristics,

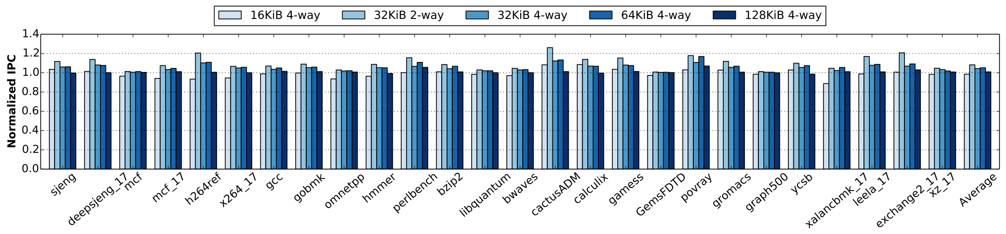


Figure 3.2: IPC with various L1 cache configurations for an OOO core, normalized to the baseline L1.

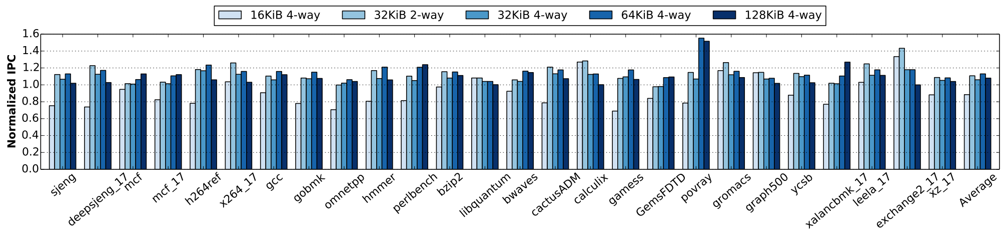


Figure 3.3: IPC with various L1 cache configurations for an in-order core, normalized to the baseline L1.

both big-data applications and the large-footprint SPEC 2017 applications are not outliers w.r.t. SPEC CPU 2006 applications.

I simulate both an OOO core with a 3-level cache hierarchy and an in-order core with a 2-level cache hierarchy. Based on the results shown in Figure 3.1, four desirable configurations are selected. The detailed parameters are listed in Table 4.1 of Chapter 4. Note that these configurations are not feasible due to the VIPT cache-indexing constrains.

In conclusion, relaxing the indexing constrains in a VIPT cache and enabling larger capacity and/or lower latency can be very beneficial. However, it is correct only if the extended bits also remain the same after address translation. Handling these extra index bits and guaranteeing the correctness properly is challenging.

3.1.1 Challenge I: Enabling Effective Cache Indexing

I propose speculatively indexed, physically tagged caches, where every cache access starts with speculatively accessing the L1, assuming that all necessary index bits will remain the same after translation. Then correctness is validated after address translation, which is done in parallel to tag matching with the VA. If the speculation succeeds, the data can be served as fast as a VIPT cache. If it fails, another access with index bits from the newly translated physical address will be issued, and in this case SIPT works in the same way as a PIPT cache. Obviously, the speculation efficiency is critical to SIPT. I propose to use a light-weighted predictor to improve speculation accuracy. Ideally, the predictor identify accesses with unchanged index bits, continues speculation with them, and furthermore, predicts the correct index bits in the PA for those accesses in which index bits are changed by translation.

3.2 Deficiency of Address Translation in Fine-grained Paging

When a translation misses in the TLBs, a page walk is required to complete translation. As illustrated in Figure 2.1, the page walk in hierarchical page tables requires multiple memory accesses. These memory accesses increase the pressure on the memory sub-system. For example, with x86 4-level page tables, one data access may require 5 memory accesses in the worst case. In addition, most caches and main memory are physically addressed, the translation must be completely finished before issuing the memory access for data. The page walk latency is on the critical path of these data accesses. It should be noticed that previous work suggests page walks typically hit in the data cache hierarchy [45]. However, because of the large gap in the latency of main memory and the caches, even the few page-walk requests that miss the caches have significant overhead [24].

3.2.1 Challenge II: Page Walk Latency

The first challenge addressed in this dissertation is the long latency of page walks. Many techniques have been proposed to improve the TLB hit rate by prefetching or coalescing TLBs [44, 55, 76, 69], but they all rely on certain access pattern of applications. Various MMU cache schemes [8, 12, 11] are used to enlarge the coverage of partial translations if a TLB miss occurs. However, even a perfect MMU cache may still suffer from the numerous accesses to the last level of the page walk. Both TLBs and MMU caches are on-chip structures and scaling them to match the growing of memory capacity is impossible. Even with large, multi-level TLBs to reduce the frequency of page walks and MMU caches to reduce the number of memory accesses per page walk, the page walk overhead can still exceed execution itself [9]. Reducing the latency of page walks, especially when a page walk access misses the last-level data cache is critical. As an orthogonal solution to reduce address translation overhead, I propose Embedded Page Translation Information (EMPTI) to embed translation information and other metadata with data. Thus, the data access and address translation can be done with only one memory access and the latency of the page walk can be avoided. However, this simple idea has one big flaw: even if we have data and translation information co-located and they can be accessed with one access, we still need to know the location first. I propose to resolve this issue by utilizing coarse-grained virtual memory schemes and employ EMPTI on top of them.

3.3 The Absence of Fine-Grained Metadata in Coarse-grained Virtual Memory

Coarse-grained virtual memory schemes such as direct segments, huge pages, and RMM can effectively reduce address translation overhead. By enlarging the granularity of address mapping, the address space can be managed with minimal amount of translation information. However, coarsening the granularity of translation sacrifices the fine-grained metadata that is available with fine-grained pages. Not having this metadata can be problematic. For instance, many libraries need to be ported if changing the page size and protection and clean/dirty status may incur large overheads when not fine enough. Even as previous work [9, 46] claims that many memory regions do not require fine-grained metadata, they rely on developers to modify code and give directives for opt-in and opt-out coarse-grained virtual memory. These directives have to be precise and absolutely correct. In addition, coarse-grained translation requires address coalescing. Because the coarse-grained mapping admits no exceptions in a single contiguous region, any conflicts need to be resolved by employing smaller regions. In summary, I conclude that the absence of fine-grained metadata in coarse-grained virtual memory reduces programmability and flexibility, introducing significant challenges in many contexts.

3.3.1 Challenge III: Coupled Translation and Metadata

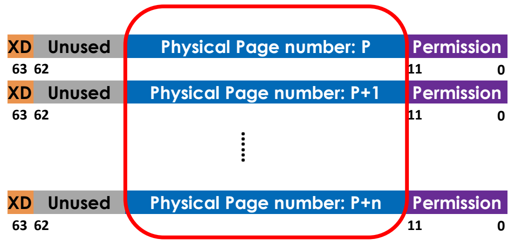
The second challenge addressed in this proposal is coupling translation and metadata. Conventionally, translation information and metadata are managed in the same structure and maintained in the same granularity. This legacy unnecessarily limits our design space and introduces the dilemma that we wish to utilize coarse-grained translation for speed but desire the programmability and flexibility benefits of fine-grained metadata. I propose to separate the granularity of address mapping and metadata management in a novel way that maintains the full richness of current metadata at fine-grained page granularity while achieving efficient translation. Specifically, a coarse-grained virtual memory (e.g. direct segments) will be used to manage address translation, but this translation is treated as a speculative hint. When accessing memory translation information in EMPTI will be checked to validate the speculation. If speculation fails, the conventional page walk will be employed to ensure the correctness. With EMPTI, metadata are always maintained in fine granularity and the per fine-grained page translation information enables gaps in mostly contiguous regions.

3.4 Metadata in the Data Cache Hierarchy

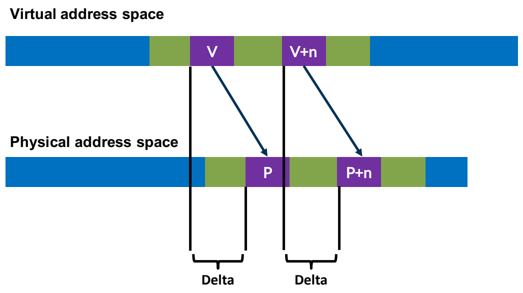
The overhead of page-table based virtual memory address translation and metadata access continues to be a significant challenge [9]. I observe that while translation-specific caching structures (e.g., TLBs and page-walk caches [8]) are very important, the regular data cache hierarchy also plays an important role in reducing the latency of address translation [64, 72]. My study in Chapter 6 shows that PTEs can take substantial LLC capacity. On an Intel Core i5-4590 processor with 32GiB main memory, PTEs take up to 4MiB out of the 6MiB LLC.

3.5 Redundancy in Metadata

Previous work suggests two types of redundancy in metadata. First, it is not necessary to keep a copy of permissions for every page because the vast majority of pages (more than 99%) use the same permissions (readable and writable) [9]. In addition, the physical address takes the largest space in a PTE (Figure 3.4a), while previous work [70] suggests that the OS naturally assigns contiguous physical pages to contiguous virtual pages (Figure 3.4b). Thus, all pages in the same contiguously-mapped region share the same delta between virtual addresses and physical addresses. By storing one delta for a contiguously-mapped region instead of physical addresses for all pages, significant cache space can be saved and more pages can be cached with the same capacity.



(a) Page table entries in a contiguously-mapped region (x86).



(b) Virtual to physical address mappings with cross page contiguity.

Figure 3.4: Natural address mapping contiguity.

3.5.1 Challenge IV: Efficient Metadata Caching

Metadata are usually small-sized supporting data to serve various functionality of the system. Yet, metadata can take a substantial fraction of cache capacity in state-of-the-art processors. At the same time, the highly redundant nature of metadata indicates opportunities for more efficient representation and storage. I propose to use Delta Caching for better caching efficiency of page tables. Delta caching modifies the PTE by replacing the physical frame number with the delta between virtual address and physical address. All pages in the same contiguously-mapped region can be represented with it. For each page, instead of storing a full 8B PTE, a 2B pointer to a separate *delta array* is stored. Because this pointer (2-byte in our design) is much shorter than 8-byte PTE, the overall cache efficiency increases.

Chapter 4

Flexible L1 Caches with Fine-grained Paging

As discussed in Chapter 3, relaxing index restrictions expands the design space of L1 caches and enables more trade-offs. In this chapter, we focus on how to make these appealing cache configurations feasible. [1]

4.1 Speculatively Indexed, Physically Tagged Caches

We proposed speculatively indexed, physically tagged (SIPT) caches to handle the situation that cache indexing bits beyond the page boundary may change during address translation. The simplest variant of SIPT always speculatively accesses the cache assuming that all necessary index bits will remain the same after translation, including those beyond the page granularity. SIPT, Like VIPT, performs address translation in parallel to accessing the cache arrays (Step 1 in Figure 4.1). For performance, all ways are accessed together so that overall latency can be reduced. After address translation, all cache tags read in Step 1 are compared with the physical addresses to

[1]This chapter is based on a published work [89], T. Zheng, H. Zhu and M. Erez, "SIPT: Speculatively Indexed, Physically Tagged Caches," 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, 2018, pp. 118-130. I proposed and evaluated the main ideas and collaborated with H. Zhu and M. Erez to accomplish this work.

select the correct cache line. At the same time, SIPT compares those index bits that were speculated with their values after address translation (Step 2 in Figure 4.1). If all speculated bits indeed are the same the "fast" access completes (Step 3). If any of the speculated bits do not match, the cache request must be repeated with the correct index bits from the PA (Step 4), slowing down the access. Fast accesses are as fast as a VIPT cache, or faster if the relaxed design constraints enable a lower-latency configuration. A slow access, on the other hand, only issues after address translation like a PIPT cache. In addition every slow access wastes energy and contends for the L1 cache port. Note that there are no coherence implications because only the L1 cache is accessed speculatively and no action (other than another access) is taken on a misprediction (tag mismatch).

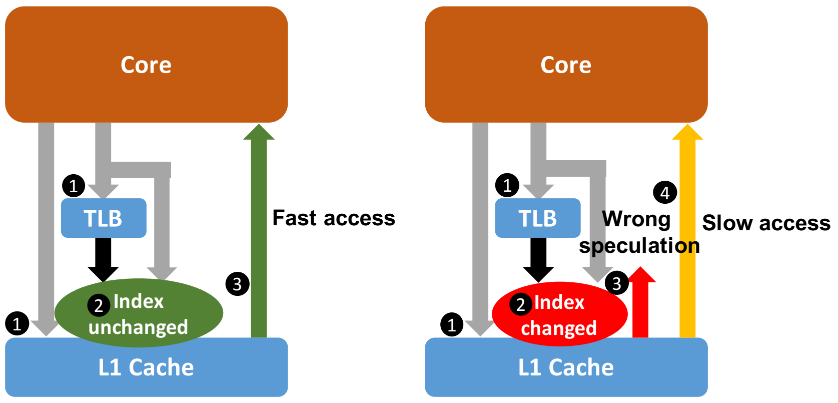


Figure 4.1: SIPT cache access when speculated index is unchanged (left) and changed (right).

4.1.0.1 Speculation Accuracy

Figure 4.2 shows the percentage of memory accesses that are speculated correctly, depending on how many speculative index bits are required. Each component of each stacked bar in the figure represents fast accesses for the number of bits required and all other accesses are slow. The most strict scenario, *hugepage* includes only those accesses from huge pages (for which 21 address bits are guaranteed not to change).[2] While some applications (e.g., libquantum and zeusmp) have most accesses targeting transparently mapped huge pages, many others have the vast majority of accesses to normal 4KiB pages. Those applications with a low correct speculation rate are likely to suffer performance degradation with SIPT compared to the VIPT baseline.

Reasonable L1 configurations, however, do not use 2MiB sets (2^{21}) and only require $1-3$ index bits beyond the page granularity. In these scenarios, the correct prediction rate is much higher overall. If only a single speculative index bit is needed (e.g., for a 32KiB 4-way L1), all but two applications (gromacs and CactusADM) have majority fast accesses.

4.1.1 Naive SIPT Performance

To evaluate the performance of this naive version of SIPT, which always speculates, we evaluate different SIPT configurations with the detailed

[2]We run Linux 3.12 with transparent hugepage management turned on and collected our traces on a system that is regularly used and which had an uptime of weeks. In the dissertation, we will include results with a fragmenter form Ingens [52] running in the background to mimic an extremely fragmented system.

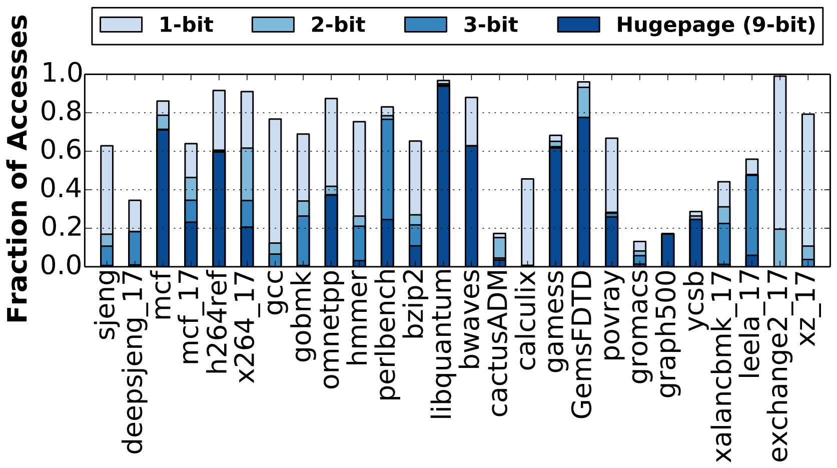


Figure 4.2: Fraction of correct speculations vs. the number of index bits that must be predicted.

parameters shown in Table 4.1, and compare to the baseline L1. We focus on OOO core and only show the results for the 32KiB 2-way SIPT configuration (with 2 extra index bits), which performs the best as an ideal cache in an OOO processor. Figure 4.3 summarizes the results and shows the IPC normalized to the baseline, and compares with the ideal cache. The figure also shows relative extra accesses due to misspeculation $\left(\frac{\text{accesses}_{SIPT}}{\text{accesses}_{baseline}} - 1\right)$.

SIPT caches with lower associativity and shorter latency achieve IPC improvement in many applications; e.g., h264ref, and perlbench, exhibit 7.3% and 8.9% IPC speedup. However, because speculative bits are used, naive SIPT suffers a high misspeculations rate. For example, in some applications, e.g., calculix and gromacs, less than 5% speculations succeed with 2 extra index bits. When misspeculations happen, SIPT generates slow accesses.

Similarly in Figure 4.4, we show the relative energy $\left(\frac{E_{\text{SIPT}}}{E_{\text{baseline}}}\right)$ of the

	In-Order processor with 2-level cache hierarchy		**Out-of-Order processor with 3-level cache hierarchy**	
Core	2-wide, in-order 3.0 Ghz		6-wide issue, OOO 192-ROB, 3.0 GHz	
TLB	L1: 64-entry, 4KiB pages; 32-entry, 2 MiB pages, 2-cycle L2: 1024-entry unified, 7-cycle			
L1	Configuration	Latency	Energy per access	Static power
	32KiB 8-way VIPT	4-cycle	0.38 nJ	46 mW
	32KiB 2-way SIPT	2-cycle	0.1 nJ	24 mW
	32KiB 4-way SIPT	3-cycle	0.185 nJ	30 mW
	64KiB 4-way SIPT	3-cycle	0.27 nJ	51 mW
	128KiB 4-way SIPT	4-cycle	0.29 nJ	69 mW
	Slow access in SIPT starts right after TLB access			
L2	None		256 KiB, 8-way, 12-cycle, private, 0.13 nJ per access, 102 mW static power	
LLC	1 MiB, 16-way, 20-cycle, shared, 0.29 nJ per access, 532 mW static power		2 MiB 16-way, 25-cycle, shared, 0.35 nJ per access, 578 mW static power	
DRAM	8-bank, 4-channel, DDR3, 16 GiB total			
Note	LLC size increase proportional to core count for multi-core evaluation.			

Table 4.1: Simulated system configurations

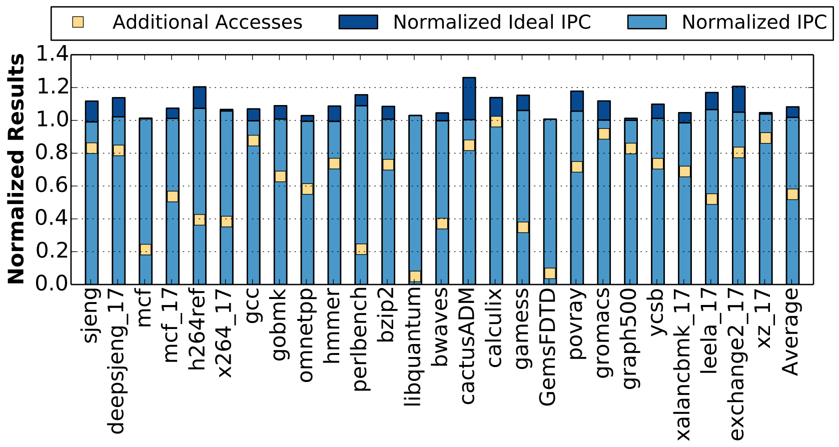


Figure 4.3: IPC and additional L1 accesses with a naive SIPT 32KiB/2-way/2-cycle cache for an OOO core normalized to the baseline L1.

whole cache hierarchy and also compare with the ideal cache. Some applications such as libquantum and GemsFDTD exhibit energy savings close to ideal. However, many other applications exhibit significant gaps between naive

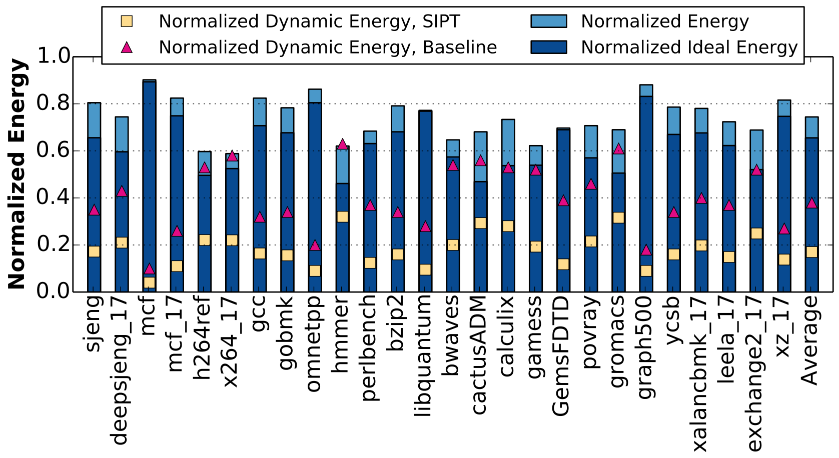


Figure 4.4: Cache hierarchy energy of naive SIPT 32KiB/2-way/2-cycle for an OOO core normalized to baseline.

SIPT and ideal. On average, naive SIPT reduces total cache energy to 74.4%, which is 8.5% worse than ideal. We also show the relative dynamic energy $\left(\frac{E_{\text{dynamic}}}{E_{\text{baseline\_total}}}\right)$ for both SIPT and baseline. SIPT reduces dynamic energy significantly.

Overall, due to a high misspeculation rate, the naive SIPT is far from ideal. We now aim to both reduce extra L1 accesses and increase the number of fast accesses.

4.2 Predicting and Bypassing Misspeculations

In this section we evaluate a light-weight predictor that determines whether a fast access is likely to succeed, and speculation should proceed, or whether cache access should wait until after translation. Our goal is to make a speculate/no-speculate binary decision and we take inspiration from the rich literature on branch direction prediction. Specifically, because we seek a light-weight predictor that can be used during instruction fetch, we evaluate a small Perceptron predictor [43].

We base our Perceptron predictor design (Figure 4.5) directly on the smallest global-history configuration proposed by Jimenez and Lin [43]. We add a global history register $x_1x_2...x_h$ that tracks the last h speculation outcomes as ones and zeros (fast access success or extra cache access failure). The predictor itself has 64 entries each being a perceptron of $h + 1$ weights $w_0w_1...w_h$. We use the memory operation program counter (PC) to index the 64-entry predictor table. Because we only use the PC, the prediction can be overlapped with other pipeline stages.

Perceptron calculates a prediction (y) by performing a dot product of the history and the weights of a specific entry in the table plus a learned bias: $y = w_0 + [x_1x_2...x_h] \cdot [w_1...w_h]$. If y is positive, we predict the index will not change and will continue with a fast access using the speculative index. If y is negative we bypass speculation and wait for the physical address before accessing the cache. Other than the smaller number of entries, all details, training algorithm, and other parameters precisely follow those of Jimenez and Lin [43] and we do not describe them.

We estimate the overhead of this perceptron predictor at just 624B of storage and a small amount of logic (6b weights, 13 weights per perceptron, 64 perceptrons). We model perceptrons as RAM with Cacti [36] (Table 3.1). The dynamic energy for reading a perceptron is only 0.34% of a baseline L1 cache access. The static power is only 0.0007% of the baseline L1 cache. Song et al. [34] suggest a 32-bit integer addition consumes $\frac{1}{10}$ the energy of reading 32 bits from a register file. Since $x_1x_2...x_h$ are ones and zeros, $y = w_0 + [x_1x_2...x_h] \cdot [w_1...w_h]$ is essentially adding $h+1$ (13 in our implementation) 6-bit integers and therefore estimated to consume less energy than reading the perceptron. Training consumes similar energy. This predictor introduces no extra latency and only negligible area and energy overheads.

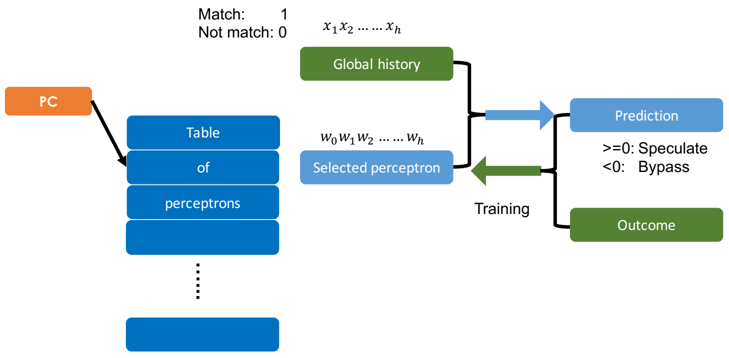


Figure 4.5: Perceptron-based predictor.

We evaluate the predictor by considering four possible outcomes. If the speculated bits are unchanged by translation and the predictor decides to speculate, we call it *correct speculation*. If the speculated bits are changed by translation and the predictor decides to bypass, we call it *correct bypass*. If the speculated bits are unchanged by translation but the predictor chooses bypass, an opportunity for a fast access was squandered, we call this *opportunity loss*. Finally, if the speculated bits are changed by translation and the predictor chooses to speculate, an extra access is generated, we call this *extra access*.

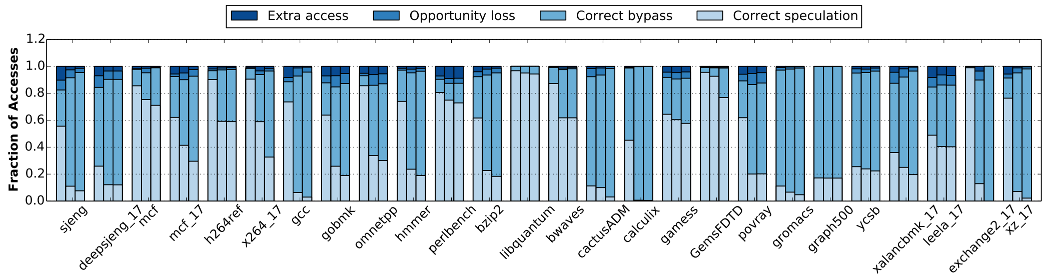


Figure 4.6: Break down of prediction results into the four possible outcomes; each group of 3 bars represents 1, 2, and 3 speculative index bits (from left to right, respectively).

This simple perceptron predictor achieves more than 90% accuracy in all applications; in fact most applications have far fewer than 5% extra L1 accesses and negligible opportunity loss (Figure 4.6). We also evaluated the sensitivity of the predictor parameters such as increasing the number of perceptrons and increasing the history length. Our experiments did not show strong sensitivity to these parameters, most likely because the prediction rate is already high. Note that we do not warm up the predictor and the results include all mispredictions. We also evaluated the sensitivity of the predictor parameters such as increasing the number of perceptrons and increasing the history length. Our experiments did not show strong sensitivity to these parameters, most likely because the prediction rate is already high. We do not warm up the predictor and the results include all mispredictions. We also evaluated various counter-based predictors, but their average accuracy is only $\sim 85\%$ and not consistent across applications. We omit the results for brevity and because the perceptron already has low overhead.

4.3 Partial Index Prediction

Intuitively, if the speculation bypass predictor is so accurate at predicting whether speculative indexing can proceed, perhaps it can be extended to predict the actual post-translation index value. Consider an SIPT design that requires a single speculative index bit. In this case, if the bypass predictor predicts to bypass, it is in effect indicating that the speculative bit is most likely not remaining the same. Therefore, flipping the bit in these cases will lead to the correct post-translation index. Because the prediction accuracy is so high, few extra accesses are added by this technique.

When there are multiple speculative tag bits, we need to predict their exact values, which is generally hard because with 3 speculative bits, it is likely that they may take any of 8 possible values. This requires a complex predictor or resulting many misspeculations. However, in the context of SIPT, predicting values is doable because of spatial locality in memory address mapping. Prior work [47, 9] establishes that memory addresses are usually mapped in coarse-grained blocks even without considering huge pages. And Pham et al. [70, 69] suggests the spatial contiguity between the virtual page numbers and physical page numbers.

Linux manages free pages using the buddy algorithm. Free pages are grouped into $1, 2, 4, \ldots 1024$ contiguous page frames and page groups of each size are then stored in linked lists. This scheme keeps the overhead of tracking free pages low.

Consider the scenario in which a user program allocates a large number of pages. This is common behavior when programs set up data structures during their initialization. The number of pages at fine-grained groups is unlikely to satisfy such requests. As a result, the buddy algorithm has to break large groups to satisfy bursts of memory allocation requests, which can lead to a significant amount of contiguous physical pages being mapped to a contiguous virtual address space. Other allocators, such as slab and eager paging [47], maintain contiguity explicitly. OS features such as Hugepages and page coloring also increase the occurrence of contiguously mapped memory blocks; page coloring tries to maintain the same low-order address bits between the VA and PA to maximize usage of the LLC.

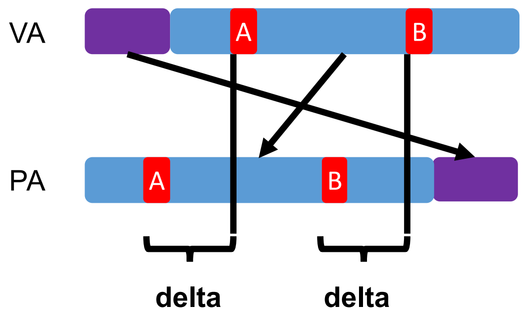


Figure 4.7: Deltas between virtual and physical addresses are constant within a single large block.

The fact that large contiguous blocks exist aids with predicting speculative index bits. For all addresses in one contiguous range (A and B for example in Figure 4.7), the delta between a virtual address and its corresponding physical address is the same. Software may even optimize for SIPT, though prediction rates are already high.

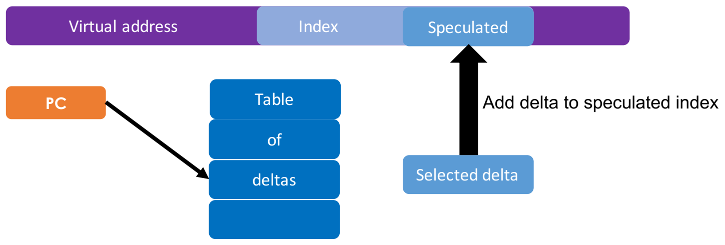


Figure 4.8: The index delta buffer predicts the delta between the speculative virtual address bits and corresponding physical address bits.

The prediction benefits from, but does not solely rely on, a coarse-grained memory mapping. Even when memory is highly fragmented and lacks multi-page contiguity, or when applications make only small allocations, deltas within each page are fixed. Thus only the first access to a page will mispredict; there are typically many L1 accesses per page.

Instead of predicting a value, we predict the VA to PA delta, which is the same for the entire range. Specifically for SIPT, only the delta of the speculative partial tag bits is required. We propose the *index delta buffer* (IDB) to predict these narrow delta values. Similar to a branch target buffer (BTB), the IDB is a PC-indexed table with each entry storing a (speculative) index delta.

In addition to updating the prediction history, we also update the expected deltas, which remain stable as long as the same regions are accessed.[3]

[3]Deltas may also change when memory is remapped explicitly with `munmap`, by copy-on-write, or on major page faults. We find that such events are very infrequent and similar analysis is included in Table 5.1

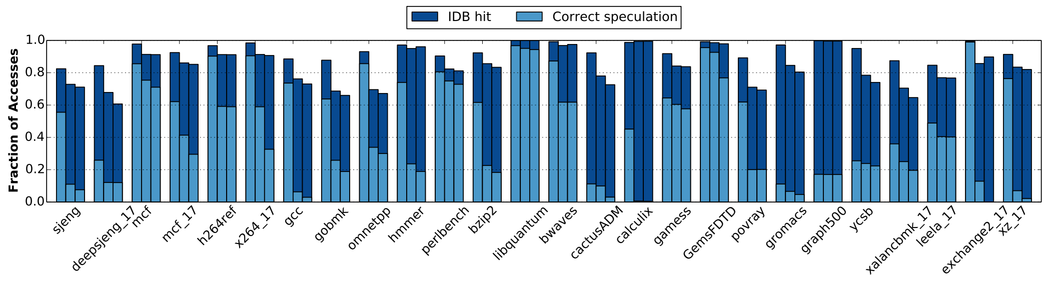


Figure 4.9: Prediction accuracy of the combined predictor when attempting to predict 1, 2, and 3 speculative index bits (the left, middle, and right bar within each group, respectively).

To compute the speculative index, we add the delta to the virtual address and truncate if it overflows. Figure 4.8 shows the design of IDB. For simplicity, we keep the same number of IDB entries as the perceptron-based bypass predictor. The storage overhead of IDB is very small because each entry is the same size as the (already small) number of speculative bits. The IDB is also accessed during fetch or decode and is off the critical path. The predicted 3-bit delta is added to the VA after address generation. The latency of the 3-bit add, which does not propagate the carry, should not increase cycle latency.

4.3.1 Combined Speculation Bypass and Index-Bit Value Prediction

Our overall index-bit predictor proceeds in two stages. First, the perceptron predictor is queried. If perceptron predicts to speculate, the speculative index is used immediately. If perceptron predicts to bypass speculation, the IDB is queried and its predicted index bit values are used to access the cache with a speculative index. Like naive SIPT, this combined predictor always accesses the L1 before translation. As long as IDB predicts correctly, slow accesses are converted to fast ones. However, because we more aggressively access the L1, more extra accesses are likely. When there is only one speculative index bit, we do not use the IDB and follow the intuitive reversed prediction technique explained earlier.

Figure 4.9 presents the accuracy and effectiveness of the combined speculation bypass and IDB predictor. We consider three possible outcomes. The first is are correctly-speculated fast access by the bypass predictor (in which case the IDB is not accessed). The second is the fraction of accesses that were predicted to bypass speculation and for which the IDB predicted the correct speculative index bit values (*IDB hits*); these would be slow accesses without the IDB that are fast accesses with it. All remaining accesses are slow and generate extra L1 accesses. Note that we also label as IDB hits those fast accesses that use the reversed bypass prediction. As with the perceptron bypass predictor, IDB consumes little area and power. We estimate the total overhead of the combined predictor at $< 2\%$ of L1 cache area and energy.

When only a single bit value needs to be predicted, over 90% (and usually close to 100%) of all accesses are fast accesses. This is in contrast to the speculation bypass predictor alone, which leaves up to 80% of accesses waiting for address translation to complete; all seven applications with low speculation rate, now have majority fast accesses (e.g., CactusADM and gromacs both go from under 20% fast accesses to more than 95% fast accesses). With 2 and 3 speculative index bits, the combined predictor successfully convert many slow accesses into fast ones. For example, gcc, calculix and xz_17 exhibited nearly no fast accesses with the bypass predictor because of poor locality between virtual and physical address. With the IDB, however, more than 70% of accesses are fast.

Figure 4.10 shows that SIPT with IDB approaches the performance of the ideal cache. The IDB enables many more fast accesses and the slow L1 accesses do not significantly hamper performance. Overall, the 32KiB 2-way SIPT cache (with 2 speculative index bits) achieves an average (harmonic mean) of 5.9% IPC speedup, only 2.3% away from ideal. In some applications (e.g., h264ref, cactusADM, calculix, leela_17, exchange2_17, and gromacs), SIPT shows more than 10% performance improvement and never underperforms baseline.

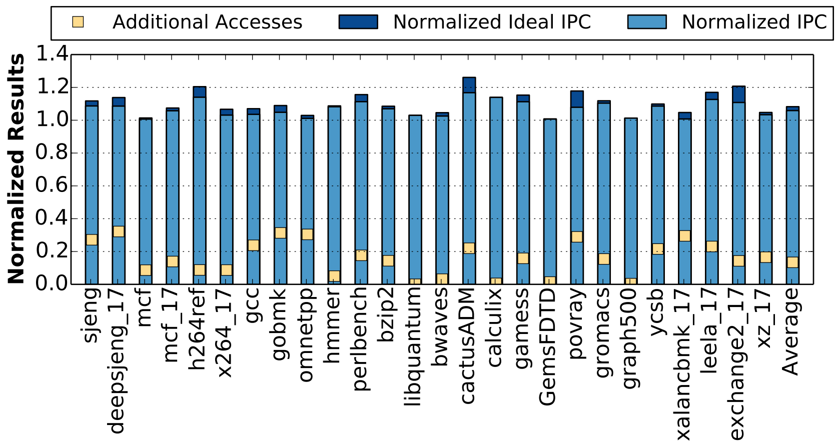


Figure 4.10: IPC and additional L1 accesses with a 32KiB/2-way/2-cycle SIPT cache with IDB for an OOO core normalized to the baseline L1.

Figure 4.11 tells a similar story for energy that the SIPT with combined predictor approaches the efficiency of an ideal cache. The energy numbers are a bit further from ideal (2.4%) than speedup because of the extra L1 accesses generated by the aggressive index-bit value speculation.

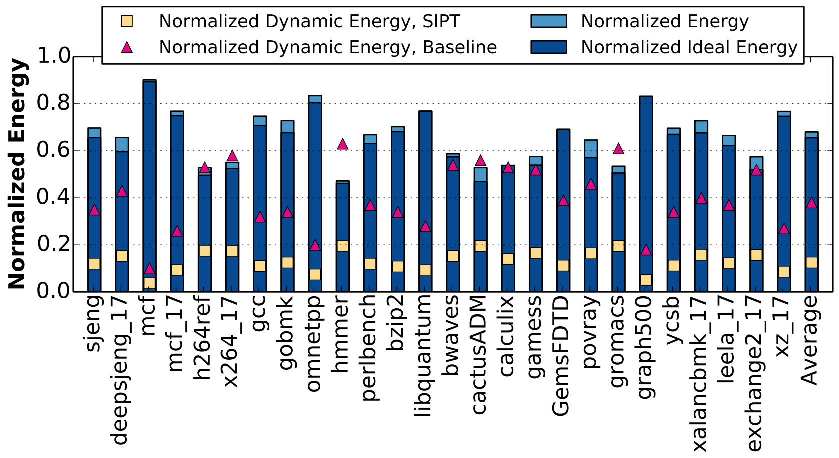


Figure 4.11: Cache hierarchy energy with a 32KiB/2-way/2-cycle SIPT+IDB for an OOO core normalized baseline.

4.3.2 Multicore Evaluation

We focus on single-core evaluation because the L1 is so tightly integrated with the core. We also evaluate SIPT in a multicore system. We simulate a quad-core processor and also quadruple the capacity of the last-level cache. We construct 11 multi-programmed workloads by mixing applications used in the single-core evaluation; every application is used at least once and the workloads are listed in Table 4.2. We recycle traces until the last core completes its initial trace to maintain a consistent level of resource contention. We report sum-of-IPC speedup, which is a simple metric that captures overall throughput improvement (speedup is relative to the multicore with the baseline cache, not a single-threaded baseline).

Mix0	h264ref, hmmer, perlbench, povray
Mix1	mcf, gcc, bwaves, cactusADM
Mix2	gobmk, calculix, GemsFDTD, gromacs
Mix3	astar, libquantum, lbm, zeusmp
Mix4	mcf, perlbench, leslie3d, milc
Mix5	h264ref, cactusADM, calculix, tonto
Mix6	gcc, libquantum, gamess, povray
Mix7	sjeng, omnetpp, bzip2, soplex
Mix8	graph500, ycsb, mcf, povray
Mix9	mcf_17, xalancbmk_17, x264_17, deepsjeng_17
Mix10	leela_17, exchange2_17, xz_17, xalancbmk_17

Table 4.2: Multi-programmed workloads.

Figure 4.12 examine SIPT for an OOO quad core. The most obvious difference between the two sets of results is that using application mixes decreases the variability in SIPT impact between different workloads. This is expected. Also as expected, using a multicore does not change any of the conclusions about the benefits of SIPT for private L1 caches.

The 32KiB 2-way cache performs the best of all configurations. Overall, the average IPC improvement is 8.1%, slightly better than we observed with a single core. The reason for the increased speedup is larger pressure on the LLC and main memory, such that L1 cache performance has a greater role. In fact, individual application speedup on each core is nearly-identical to the single-core experiments. We expect this because there is no sharing and no contention in this multiprogrammed environment.

SIPT is also able to reduce the total cache energy, but to a smaller degree than with a single core. We attribute this to the longer overall run times resulting from interference at other levels of the memory hierarchy; the static energy component is relatively larger in the multicore so the impact of SIPT is lessened.

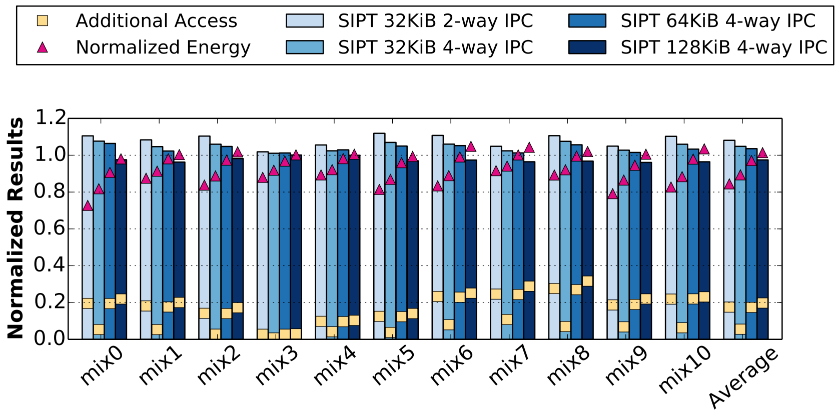


Figure 4.12: IPC, extra L1 accesses, and cache hierarchy energy of SIPT with IDB for an OOO quad core; IPC and energy normalized to the baseline L1 cache.

4.4 Further Discussion

4.4.1 Way Prediction

Way prediction [37] saves access energy compared to set associative caches where all cache lines in the same set are fetched in parallel. By predicting the data location within the cache set, only the predicted cache line is fetched. If the prediction is correct, no additional access latency is introduced. When the prediction is wrong, a second access is required to search the remaining cache lines in the set.

We evaluate the simple way prediction mechanism described in [37] that the MRU way in a set is always predicted. A small amount of metadata (3 bit

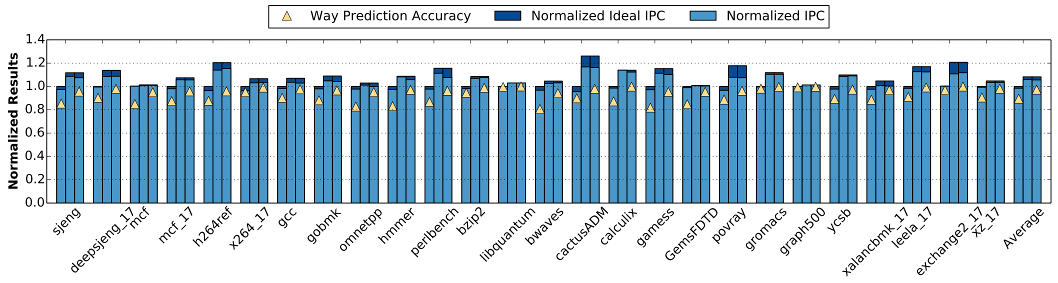


Figure 4.13: IPC normalized to the baseline L1 and way prediction accuracy. Each group from left to right: baseline L1 with way prediction, 32KiB/2-way/2-cycle SIPT with IDB, and 32KiB/2-way/2-cycle SIPT with both IDB and way prediction.

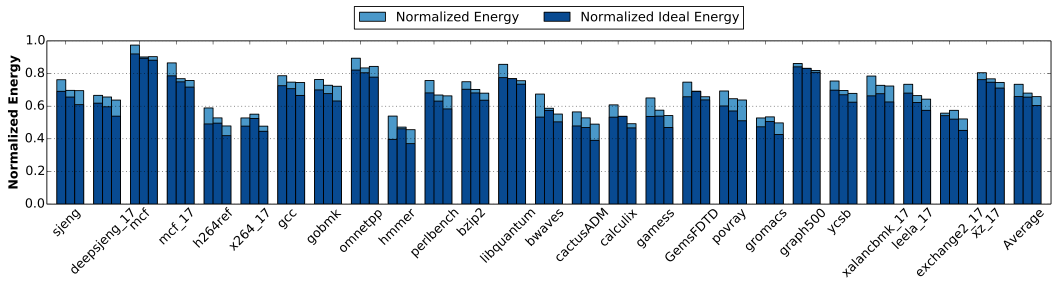


Figure 4.14: Cache hierarchy energy normalized to the baseline L1. Each group from left to right: baseline L1 with way prediction, 32KiB/2-way/2-cycle SIPT with IDB, and 32KiB/2-way/2-cycle SIPT with both IDB way prediction.

per set for an 8-way set associative cache) is accessed before cache is accessed. Although, fancy predictors may increase the accuracy of way prediction, we find that the accuracy of this simple predictor is already high and robust across applications. Unlike SIPT, way prediction requires the virtual address, thus cannot be fully overlapped with early pipeline stages. Employing more complex metadata may introducing extra latency to cache accesses. We stay with this simple prediction mechanism and optimistically modeled no extra latency for accessing way prediction metadata.

Because way prediction and SIPT are complementary, we apply way prediction to both the baseline and our 32KiB/2-way/2-cycle SIPT cache with combined speculation bypass and IDB prediction. The results are shown in Figure 4.13. And Figure 4.14 shows the normalized cache energy. In addition to being ideally indexed, ideal caches also assume way prediction always accesses the correct way. We model the energy of way prediction by reducing the relative dynamic energy according proportionally to the associativity (e.g., one-eighths of dynamic energy consumed in the baseline 8-way cache on a way prediction hit).

When applied to the baseline cache, way prediction achieves 89% accuracy and reduces cache energy by 24% on average. However, the remaining misses still reduce performance by 2% overall. When applied on top of SIPT, because of reduced associativity (from 8 in baseline to 2 in SIPT), the way prediction accuracy increases to 97.3% on average, and there is only a 0.3% performance drop compared to SIPT alone. At the same time, it saves 2.2% additional cache energy (compare to SIPT alone), superior than ideal way prediction. By reducing associativity to 2, SIPT alone already saves significant cache energy, and there is only limited space for further reduction. The saving from applying way prediction on top of SIPT is very stable among all applications because of the robust and high accuracy.

4.4.2 The Predictability of Partial Index Bits

The efficiency of SIPT relies on the accuracy of partial index-bit prediction. We evaluate SIPT with a regularly used machine that has an uptime of weeks. However, this is not necessarily the worst condition an application faces. We now discuss a few sensitivity studies with more severe operating conditions, including running applications with artificially highly fragmented physical memory and with Linux's transparent huge page mechanism turned off (thus only fine-grained 4KiB pages). We also include the impact of these parameters on the performance of the in-order machine Table 4.1.

Fragmented Memory. On a long-running system with a large number of co-running applications, the physical memory may be fragmented. When running applications on fragmented physical memory, the lack of physical memory contiguity may decrease the predictability of partial index bits. We use a tool from Kwon et al. [52] to fragment memory and quantify fragmentation using the *unusable free space index* [31], a value between 0 (unfragmented) and 1 (highly fragmented). Importantly, this index does not represent lack of memory, but rather an inability to satisfy large contiguous requests. It can be calculated with $F_u(j) = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}$ where $TotalFree$ is the size of the free space, 2^n is the largest allocation that can be satisfied, j is the order of the desired allocation size (2MiB for THP), and k_i is the number of free page blocks of size 2^i. We maintain an unusable free space index > 0.95; an extreme level of fragmentation at nearly all times and not representative of typical operation. Again, we never run out of physical memory in any experiment.

To evaluate SIPT under highly fragmented physical memory, we repeat the same simulations we conducted before with traces collected under this extreme condition.

4**KiB Pages.** Most Linux distributions enable transparent huge pages (THP) by default. We are aware that under certain circumstances, THP hurts performance [28] and should be disabled but believe this is not the general case. However, an interesting question is what impact disabling THP has on SIPT prediction accuracy and performance. Similarly, we repeat the same simulations with THP disabled, which forces all pages to 4KiB.

Removing $>$ 4**KiB Contiguity.** A more challenging (but unrealistic) condition is to force zero contiguity beyond 4KiB pages such that all 4KiB pages are mapped with different deltas. This essentially eliminates all benefits from contiguous memory mapping, thus IDB only works for references within the exact same page. We force this in simulation by tracking the page number of the last access for each IDB entry. We then apply delta prediction only if the same page is accessed and choose a random delta if a different page is accessed; this mimics zero contiguity without modifying the OS. Note that this scenario presents locality and randomness that exceed that of any reasonable system and dynamic VA to PA remappings.

Figure 4.15 shows the average IPC, cache energy normalized to the baseline, and the prediction accuracy (the percentage of *correct speculation* and *IDB hit*) for all four SIPT configurations on both OOO and in-order cores. As expected, running with fragmented physical memory or disabling THP does degrade the behavior of SIPT. However, the degradation is not significant. For instance, with a 32KiB 2-way SIPT cache (2-bit speculation) on OOO core, the prediction accuracy drops from 86.7% to 84% when running with fragmented physical memory, 83.1% when THP is disabled, and 73% when no $>$ 4KiB contiguity. The IPC improvement also drops from 5.9% to 5.3%, 4.8% and 3.8%, and cache energy increases from 67.8% to 68.3%, 70% and 71.2%. A similar trend can be observed in all SIPT configurations on both OOO and in-order cores. As mentioned in Section 4.3, our prediction mechanisms benefits from contiguous memory mapping but does not solely rely on it. Using extremely fragmented memory or disabling THP has limited impact.

4.4.3 Implications for Instruction Schedulers

Modern processors commonly support speculative scheduling to enable back-to-back execution of dependent instructions [68, 49]. The speculation relies on deterministic instruction latencies with support for rare instruction replay when latency varies for certain instructions. Load instruction latency is variable because of cache misses and way prediction and SIPT introduce another source of variability. As suggested in [49], an ideal selective replay mechanism (replay only necessary instructions) is not feasible for wide issue

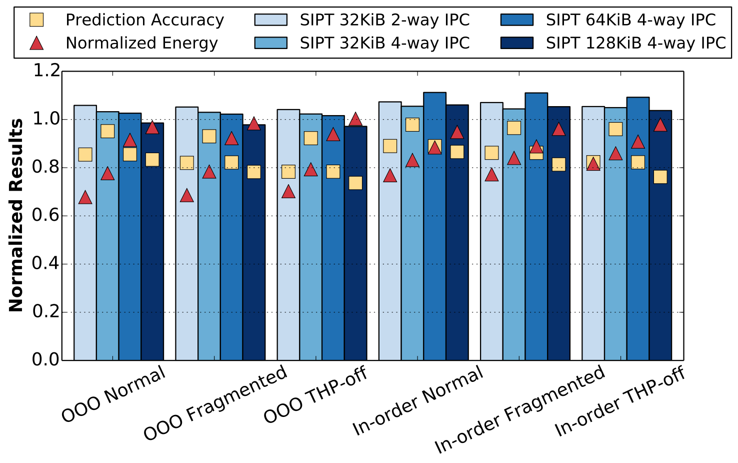


Figure 4.15: IPC, cache hierarchy energy, and prediction accuracy on OOO and in-order core with various operating conditions; IPC and energy normalized to the baseline L1.

OOO processors and replay mechanisms require trading off design complexity (HW resources) and accuracy (performance impact). SIPT is very accurate and can use existing speculative-scheduling approaches to recover from its rare mispredictions, which are a fraction of cache misses that are already addressed by the scheduler. Impact on the scheduler can be further reduced because SIPT has a built-in confidence estimator. Similarly to prior designs, expensive selective-replay resources can be reserved for more challenging loads [68, 67, 49]. Loads that SIPT predicts their speculated bits to be unchanged, an alternative simpler replay mechanism with larger penalty can be used instead of selective replay. In many applications (e.g. libquantum, zeusmp) nearly all loads do not require selective replay.

Chapter 5

Low-overhead Translation with Fine-grained Metadata

As already discussed in Chapter 3, a page walk will suffer long latency when the translation misses in the cache hierarchy and requires DRAM accesses. Previous work attempts to reduce the overhead of translation by increasing the granularity of both translation and metadata. In this chapter, we introduce a novel mechanism to achieve low-overhead translation without compromising metadata granularity.

5.1 Embedded Page Translation Information

The idea behind Embedded Page Translation Information (EMPTI) follows a simple insight: in the vast majority of cases, a load instruction that requires DRAM access for address translation also misses in the processor's cache hierarchy (Figure 5.1). In such a situation, if both the translation information and data are located in the same memory block and can be served with just a single DRAM access, the page walk effectively takes zero latency.

When the memory access for a page walk is required, we first consult a coarse-grained translation mechanism to get a physical address (the preferred

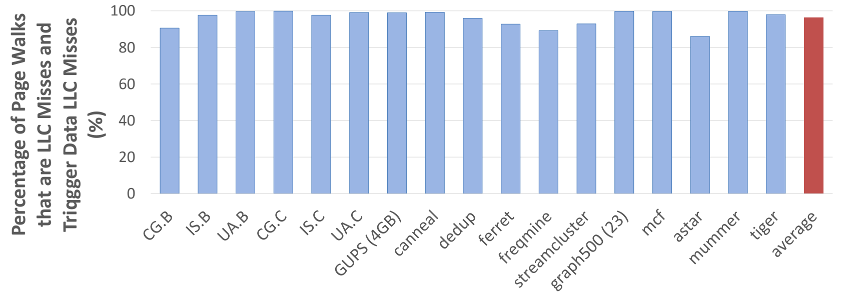


Figure 5.1: Percentage of page walks that are last-level-cache (LLC) misses which also trigger data LLC misses. Results are for a simulated quad-core system with a 64-entry L1 TLB per core, a shared 1024-entry second level TLB, an 8MiB unified LLC, and 8GiB main memory.

location), and then speculatively access that location. Before inserting data into the last-level-cache (LLC) we use the translation information to verify that we indeed accessed the correct location.

We refer to the information co-located with data, which includes a representation of the full virtual address and the permissions and access information, as an *embedded PTE (EPTE)*. When the data and translation metadata return from DRAM, EMPTI verifies that the physical location accessed matches the desired virtual address and that the access is valid (Figure 5.2). In the most aggressive variant of EMPTI, we embed the EPTE *within each memory block* and can thus complete both the data fetch and TLB fill in one DRAM access. Note that we choose a single process to be *promoted* to use EMPTI; naturally, this is the process that requires very large memory capacity.

We include the permission and other access information explicitly in each EPTE, but represent much of the address information implicitly. For

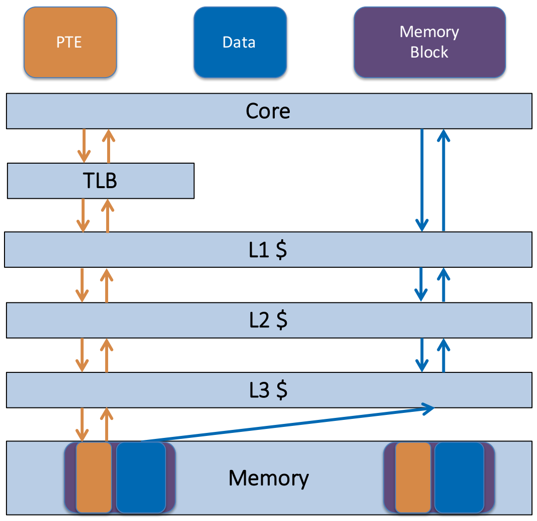


Figure 5.2: Page walk and data access served by a single memory access with EMPTI. The light and dark portions of a memory block represent the embedded PTE (EPTE) and data portions, respectively.

example, even in a small x86 system with 4GiB physical memory and 4KiB pages, each virtual page number (VPN) is represented by 36 bits (48 bits of usable virtual address minus 12 bits of offset [41]) and 20 of those bits (2^{20} = 4GiB) are implicit in the physical address chosen as the preferred location (more bits are implicit when physical memory is larger than 4GiB). Thus, each EPTE requires a maximum of 30 bits: 16 bits for the address tag (fewer with larger physical memory), 11 bits for the permissions and access bits that include the execute/disable (XD) bit, and a valid bit; the valid bit is necessary because the preferred location may sometimes be allocated to a different process. Figure 5.3 depicts an EPTE.

Note that we do not store the accessed and dirty bits in an EPTE because those bits are sticky in the actual PTE and will never be reset by hardware [41]. Phrased differently, because all the information communicated by hardware to software does not depend on a particular initial state, there is no need to store it in the EPTE and only the PTE must be updated.

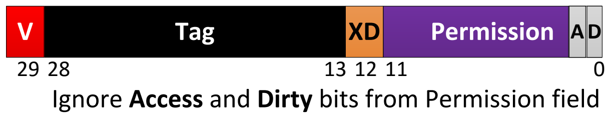


Figure 5.3: Embedded Page Table Entry (EPTE).

Figure 5.5 illustrates how a load instruction from the promoted process, which misses the TLB, accesses memory. The VPN is used to generate the speculative preferred physical address. The corresponding memory block and its EPTE are then read into the processor. The tag in the EPTE is combined with the physical address bits that implicitly match the preferred location and compared to the virtual address of the load. If the two match and the EPTE is valid, the permission bits are checked as usual and the TLB is filled from the EPTE. An EPTE may be invalid because the OS may choose to ignore the Embedded Page Translation Information optimization and map a page to a physical address that is not its preferred address. In such a case, the valid bit is unset to indicate that the regular TLB procedure should be followed.

Preferred Address Mapping. For evaluation and explanation, we use the lower-significance bits of the VPN as its preferred physical address for EMPTI;

this straightforward mapping works well for targeted Figure 5.4 HPC applications. Note that EMPTI can be generalized by combining it with other mapping schemes, for example, if direct mapping yields a high conflict rate, direct segments or redundant memory mappings. In a combined architecture, addresses are mapped in coarse granularity (segments or ranges), however, metadata is always kept in fine granularity with EPTEs.

Cache Interactions. With EMPTI, both data and its translation metadata are read together. Any EPTE access also fetches data, even if the data is already in the processor cache. Therefore, the cache is queried using the preferred physical address and if the data is already in the cache, the cache fill is canceled and only the EPTE is forwarded to the MMU. Canceling the data fill is rarely needed, as shown in Figure 5.1.

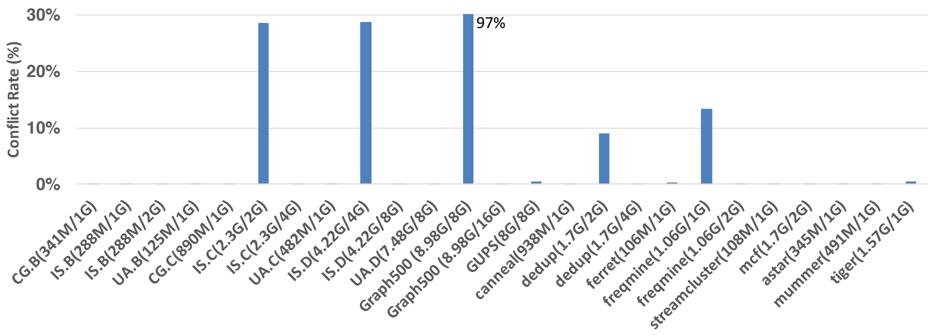
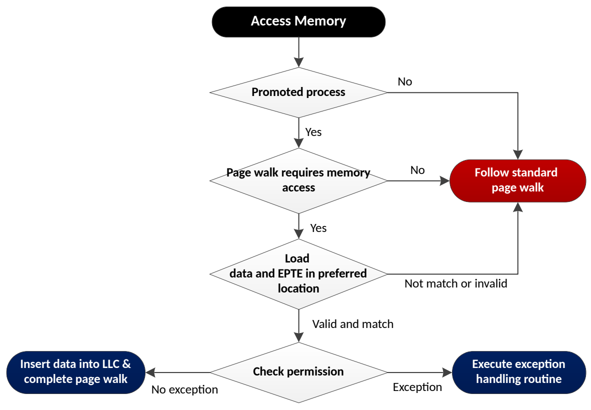


Figure 5.4: Direct mapped conflict rate with various memory capacities (footprint/memory).



Access Memory
Promoted process
No
Yes
Page walk requires memory access
No
Follow standard page walk
Yes
Load data and EPTE in preferred location
Not match or invalid
Valid and match
Check permission
Insert data into LLC & complete page walk
No exception
Exception
Execute exception handling routine

Figure 5.5: Application memory access with Embedded Page Translation Information.

5.1.1 EPTE Storage

The key feature of EMPTI is that the memory controller accesses both the data and its EPTE at the same time. To support this using standard cache line-granularity memory access, we embed the EPTEs within memory in a way visible only to the memory controller. I discuss several embedding options below as well as the architectural modifications needed for accessing the EPTEs. The embedding options offer different tradeoffs in complexity and performance. All options are implementable with the standard memory system designs used by commodity processors.

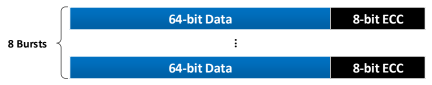
ECC Embedding. In the most aggressive forms of Embedded Page Translation Information, an EPTE is embedded withing each memory block. This requires identifying 32 bits of EPTE storage in each 512-bit memory block.

We describe how some of the ECC bits used for memory protection may be safely repurposed for EPTE storage. While some prior work has used ECC bits for various purposes, to the best of our knowledge there have not been proposals that use more than a handful of bits and for which an evaluation of the impact on reliability has been published.

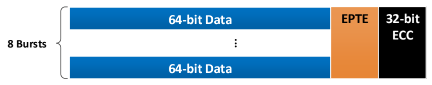
A standard ECC DIMM has 12.5% redundancy and uses a 72-bit wide channel where 64 pins are used for data and 8 for ECC information. Thus, each 512-bit memory block has 64 bits of redundant storage. Recent work by Kim et al. [51] has proposed a single-pin-correct triple-pin-detect (SPC-TPD) ECC code that uses just 32 bits for each 512-bit memory block, yet has equivalent correction capabilities to current SECDED ECC and detection coverage similar to chipkill-level ECC. This is possible because SPC-TPD treats an entire memory block as one long codeword, rather than forming codewords from individual (or paired) bus beats as done with other designs [42, 22].

With just 32 bits needed for ECC, the 32-bit EPTE can be effectively stored within each memory block (Figure 5.6). Note that SPC-TPD was designed for a 64-bit wide data channel with 4 bits of redundancy per bus beat but we use it to protect a 68-bit data channel after adding the 4-bit wide (with burst 8) EPTE component. We conducted detailed error analysis using the methodology described by Kim et al. [51] and conclude that the expected error rate increases by just 6% compared to that reported for the original SPC-TPD design; this small difference has no practical impact on reliability considering the already very low memory failure and error rates with ECC enabled.

Accessing EPTE Storage. The translation hardware only reads EPTEs. This is done at the memory controller by forwarding EPTE information to the MMU. Software must populate EPTEs as we discuss in the following section. This can only be done by the operating system and requires a new privileged `store-EPTE` instruction. This instruction accepts the physical address into which the EPTE should be embedded and the 32-bit EPTE value. The instruction then performs a read-modify-write on the physical address to embed the EPTE, and encodes the new ECC symbols for protecting both the data and the EPTE. It may also be beneficial to provide a privileged DMA operation that sets EPTEs as it copies data, which is useful when a new frame is allocated.



(a) Original data allocation with ECC DIMM



(b) Data allocation when storing EPTE with ECC redundancy

Figure 5.6: Storing EPTE with ECC redundancy

DRAM Row-Buffer Embedding. With row-buffer embedding, EMPTI does not attempt to retrieve both the data and EPTE in a single memory access. The performance impact of two separate memory accesses is kept low by locating the EPTE within the same DRAM row buffer as the data. The EPTE information for every memory block within a single memory frame is the same. Thus, it is not necessary to replicate the EPTE. We do not wish to expose the EPTE to software directly, just as with the ECC-based embedding. We therefore augment the hardware address mapping logic to account for EPTE locations. We do so in a way that maximizes the likelihood that both data and its EPTE are mapped to the same DRAM row.

We show an example of this mapping for a system with 4KiB frames and 8KiB row buffers in Figure 5.7. In this configuration, 2 EPTEs are stored in the last 64B memory block of each DRAM row. Thus, one frame fits entirely in the DRAM row along with its EPTE, while the second frame has one memory block mapped to a different DRAM row. On average, 127 of every 128 data blocks and their EPTE can be accessed with a single DRAM row activation command (if necessary) followed by two DRAM column read commands. The two reads are issued back-to-back and thus the latency of translation is made negligible, though it still incurs some bandwidth overhead. One memory block out of every two frames is mapped to a different DRAM row and accessing it requires higher overhead, or EPTE access can simply be skipped. The mapping also works with larger virtual memory pages where only a single EPTE is stored in each memory row.

This technique has a low storage overhead of just 512 bits for every 8KiB DRAM row, or $\sim 0.8\%$. Its implementation overhead is also low because it only requires modifying the DRAM address generation logic in the memory controller as also discussed by Chou et al. [17]. The last memory block of every two frames requires a more complex, though still deterministic, address translation, which can be accomplished with simple low-cost logic.

Note that this general idea of row-buffer embedding has been first proposed for ECC storage [21, 33, 88] and it is commonly believed a variant is in use for DRAM ECC protection in some NVIDIA GPUs.

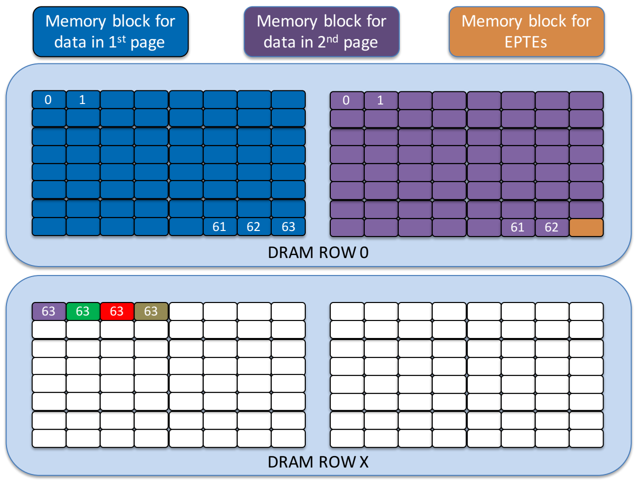

Memory block for data in 1st page
Memory block for data in 2nd page
Memory block for EPTEs
0
1
61
62
63
0
1
61
62
DRAM ROW 0
63
63
63
63
DRAM ROW X

Figure 5.7: Visual representation of mapping data and EPTE storage to DRAM rows.

5.1.2 Software Support

Software plays a role in using EMPTI as it decides which process and pages to promote to use EMPTI and then maintains EPTEs. Note that the focus of this paper is on the EMPTI mechanism itself and its impact on performance, and we only outline the necessary software changes. We start with describing how memory is allocated and deallocated for the process that is promoted to use Embedded Page Translation Information before discussing EPTE maintenance and process promotion and demotion.

Memory Allocation and Deallocation. The allocation process with EMPTI is illustrated in Figure 5.8. When the OS allocates a frame, it first checks whether the process is the promoted process. For the promoted process, the OS attempts to allocate the preferred physical frame for the requested virtual page from the frame free list. This likely requires a more sophisticated data structure for tracking the free list, but we leave such a design for future work. If the preferred frame is available, it is allocated by updating the page table. The OS then issues `store-EPTE` instructions to initialize the EPTEs of the newly-allocated frame. The page table is then updated to mark the page as promoted to ensure the EPTE is kept up to date. Note that with row-buffer embedding, only a single `store-EPTE` is needed. If the preferred frame is not available, a different frame is allocated. The OS must then unset all EPTEs in that frame to prevent false positives when hardware attempts to speculatively read an EPTE. This same procedure is followed when allocating a frame for a non-promoted process.

Note that initializing the EPTEs can be done simultaneous with frame data initialization; with current OS policies, it is standard practice to initialize the data of a frame to either all-zeros or a copy of another frame (when using copy-on-write). Thus, the overhead of EMPTI is small if hardware provides a special DMA operation for simultaneously copying data and setting EPTEs.

Deallocating a single frame is straightforward. In addition to updating the page table and adding the frame to the free list, all EPTEs in the frame are invalidated. This adds overhead to deallocation that may not be present in the standard OS flow, but we show such cases are rare (other than on process termination). EMPTI also introduces overhead when a promoted process terminates because its page table must be traversed frame by frame and all EPTEs invalidated. Because promoted processes have large footprints, it is likely that the time spent invalidating EPTEs will be small compared to the lifetime of the process. These overheads are smaller with row-buffer EPTE storage.

EPTE Maintenance. Every time the OS updates a PTE, the changes are reflected to the EPTE with a `store-PTE`; an EPTE exists only if the process is promoted and the PTE is marked as having an EPTE. The OS never needs to read an EPTE because any state that is modified within the TLB is written only into the PTE itself.

EPTEs must also be updated when a page-directory entry is changed. This is because permissions are computed hierarchically with higher translation levels determining overall access control. Without Embedded Page Trans-

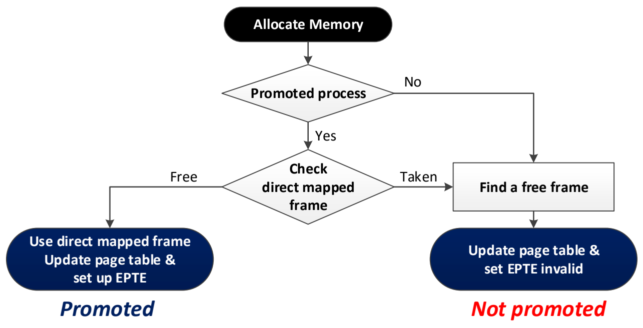

Allocate Memory
Promoted process
No
Yes
Check direct mapped frame
Free
Taken
Find a free frame
Use direct mapped frame Update page table & set up EPTE
Update page table & set EPTE invalid
Promoted
Not promoted

Figure 5.8: Workflow for memory allocation.

lation Information, by just changing a single page directory entry, the OS may change the permissions or invalidate an entire branch of the hierarchical page table. While this overhead may sound high, our analysis suggests that such changes are very rare and will not impact performance.

We analyze the potential overhead of updating PTEs by counting memory-management system calls and page faults observed when running our benchmarks as well as their run time overhead (using Linux `time` [60], `strace` [61], and LMbench [58]). We measure on a single-socket Haswell-based system and the set of benchmark applications include PARSEC [13], NAS Parallel Benchmarks [7], HPC Challenge Benchmarks [56], Graph500 [32], SPEC CPU2006 Benchmarks [35], and BioBench [4]. Figure 5.9 shows that all but one of our benchmarks spend roughly 1% or less of their time processing page faults (dedup spends 3.9%). The overall rate of page faults is small and the average number of page faults per allocated page across the entire duration of an application is typically very small (the exception is ferret with with 18.3 page faults per page). Table 5.1 shows that the total number of memory management system routines called by each application (`mmap`, `munmap`, and `mprotect`) is also small. This implies an infrequent EPTE update rate.

5.2 Evaluation Methodology

We compare the execution time of traditional 4KiB pages, Linux's transparent huge pages (THP) mechanism that use the processor's support for 2MiB pages, and Embedded Page Translation Information. The system

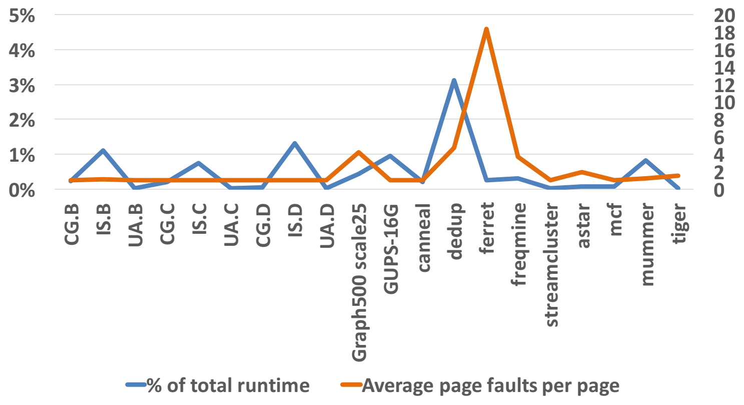


Figure 5.9: Percentage of total run time and average page faults per allocated page.

	mmap	munmap	mprotect	run time (sec)
CG (B/C/D)	28/28/28	1/1/1	21/21/21	20.37/56.04/4878.21
IS (B/C/D)	7/7/7	0/0/0	3/3/3	3.31/40.03/44.84
UA (B/C/D)	28/28/28	1/1/1	21/21/21	58.43/229.82/4498.34
Graph500 scale25	218	199	15	2252.07
GUPS-16G	4	0	3	219.44
canneal	26	2	15	63.12
dedup	35	18	22	33.33
ferret	43	27	26	92.37
freqmine	78	43	16	109.91
streamcluster	33	5	15	103.89
astar	12	11	0	133.76
mcf	14	6	4	331.12
mummer	5	4	0	8.84
tiger	69	64	0	751.57

Table 5.1: Number of `mmap`, `munmap`, and `mprotect` calls.

configuration is listed in Table 5.2. We use a performance model that relies on per-application parameters measured on real hardware (described in Section 5.2.1). We do not use a simulator. As explained in prior work, studying translation issues requires observing long execution durations of large-memory applications—something that cannot be done in a simulatorFurthermore, the details of memory and translation caching of high-end processors are both complex and proprietary; making accurate modeling effectively impossible.

We gather results for a set of memory intensive applications from a wide variety of benchmark suites, include PARSEC [13], NAS Parallel Benchmarks [7], HPC Challenge Benchmarks [56], Graph500 [32], SPEC CPU2006 Benchmarks [35], and BioBench [4]. These applications have also been used in prior research on reducing the overhead of address translation [46, 9]. The detailed parameters of each application are listed in Table 5.3. Note that we intentionally focus on high performance computing applications that have exclusive ownership of the vast majority of memory and our system is not virtualized and thus has a lower baseline address translation overhead.

Processor	Intel i5-4590 (Haswell) 4 cores, 3.3GHz
L1 Cache	32KiB instruction, 32KiB data, Private
L2 Cache	256KiB, Private
L3 Cache	6MiB, Shared
L1 DTLB	4KiB 64-entry 4-way 2MiB 32-entry 4-way, 1GiB 4-entry
L2 TLB	Unified 8-way, 1024-entry
MMU cache	L4 2-entry L3 4-entry, L2 32-entry 4-way
Main memory	32GiB
Linux kernel	3.13.0

Table 5.2: System configuration.

5.2.1 Performance Model

We directly measure the total run time and page-walk overhead for each application with both standard 4KiB pages and THP using the Intel performance counter monitors available on the Haswell microarchitecture and later generations. We estimate the run time with EMPTI using a performance model that is similar to the model introduced by prior work [46, 9]. The model

Workload	Benchmark	Input Data Set	Size (MiB)
NAS parallel bench	CG	B / C / D	404 / 979 / 16711
	IS	B / C / D	160 / 1560 / 4640
	UA	B / C / D	128 / 493 / 7296
PARSEC bench	canneal	native	2679
	dedup		4173
	ferret		2867
	freqmine		2387
	streamcluster		383
HPCC	GUPS	16GiB	16409
Graph500	Graph500	25	37070
SPEC CPU2006	473.astar	Reference	384
	429.mcf		1684
Bio Bench	mummer	Default	467
	tiger		611

Table 5.3: Application parameters.

is straightforward: EMPTI eliminates the overhead of page walks that are LLC misses and we therefore simply exclude the time necessary for those memory accesses from the total execution time. To do this we first subtract the page walk overhead measured with 4KiB pages from the total execution time measured with performance counters dedicated to these events. We then add back the estimated time of servicing page walks from the cache hierarchy, effectively excluding the memory access time. Again, we rely on performance counters and a simple additive latency model based on level of hierarchy and translation structure that serviced each request; we also assume no overlap in translation processing – this is reasonable given their relatively high latency and the fact that the processor we use has a single hardware page walker. Because we are running on MMU cache equipped processors, the impact of reducing page walk latency from caching partial translations is always included. Since a conflict generates a memory access before the followed regular page walk procedure, we conservatively estimate the performance of a conflicted access by doubling the page walk latency. The details of this model and parameters collected are summarized in Table 5.4.

Note that unlike prior work [46, 9], we measure the ideal execution time separately for 4KiB pages and THP. The reason is that some of the applications we use exhibit relatively large THP management overheads and thus have a higher ideal execution time than with 4KiB pages. This is particularly evident when running CG.D and Graph500 with THP enabled. These applications exhibit an extremely high unusable free space index, which suggests severe external fragmentation and likely increases various management overheads [31].

Row-Buffer Embedding Performance Model. The model above assumes that data and its EPTE are always fetched in a single access, as is the case with ECC embedding. However, with row-buffer embedding, two accesses to the same DRAM row are required. With commodity hardware, we

Collected Statistics	
$T_{4K/THP}$	Total execution cycles with (4K / THP)
$PW_{4K/THP}$	Cycles spent in page-walks (4K / THP)
$PL_1/PL_2/PL_3$	Page walks served by L1/L2/L3 cache (4K)

Parameters	
$C_1/C_2/C_3$	L1/L2/L3 cache latency (4/11/30 cycles)
CR	Conflict rate

Model	
Ideal execution time	$E_{4K/THP} = T_{4K/THP} - PW_{4K/THP}$
EMPTI page walk cycles	$PW_{EMPTI} = C_1 * PL_1 + C_2 * PL_2 + C_3 * PL_3$
Conflict overhead	$T_{Conflict} = (PW_{4K} - PW_{EMPTI}) * CR * 2$
EMPTI total run time	$T_{EMPTI} = E_{4K} + PW_{EMPTI} + T_{Conflict}$

Table 5.4: Performance model and measured parameters. The 4K/THP notation indicates that two parameters are measured – once when running with 4KiB pages and a second time with THP.

cannot introduce the second access with appropriate timing low-enough overhead. Thus, to estimate performance for EMPTI with row-buffer embedding, we construct a microbenchmark to estimate the worst-case latency impact of separately fetching data and its EPTE from the same DRAM row. This microbenchmark has two phases. In the first phase it accesses a long sequence of random addresses that cannot be prefetched. In the second phase, the same set of accesses is repeated but each is paired with a second access to the immediately adjacent cache line.

We first run a single-threaded instance of this microbenchmark and measure a 4% overhead for paired accesses. We then run a set of multi-threaded experiments under four scenarios with different levels of memory bandwidth requirements. The first scenario is increasing the number of threads in the microbenchmark from 1 to 4 where we measure an 8% maximum overhead. The second repeats this experiment while performing both reads and writes (50/50 ratio) and the overhead is similar. In the third scenario we run a single

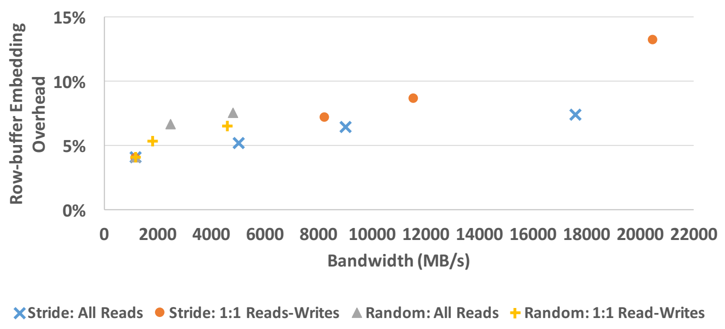


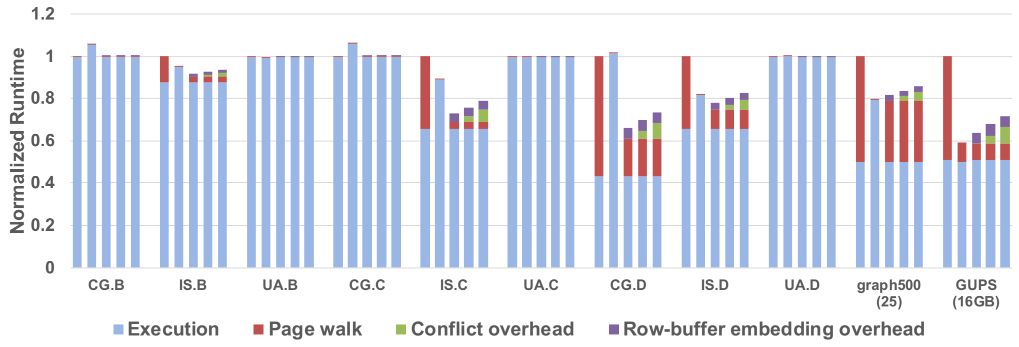
Figure 5.10: Row-buffer embedding overhead.

thread of the microbenchmark and introduce a second thread that performs strided memory reads. We change the stride to control bandwidth utilization and vary bandwidth from 1 to 18GiB/s. We measure a row-buffer embedding overhead of $4-7\%$ in this experiment with a linear relation between overhead and overall bandwidth. The fourth scenario repeats the above experiment where the bandwidth-consuming thread performs an equal number of reads and writes. In this case total bandwidth consumption is $1-21$GiB/s and overhead is $4-13\%$ (linear relation). These results are shown in Figure 5.10. Note that the maximum bandwidth measured on our system with a bandwidth stress test is 24GiB/s (theoretical peak of 25.6GiB/s).

The actual number may vary depending on the configuration of the specific machine and the policy employed by its memory controller. However, on our typical state-of-the-art processor, we conclude that row-buffer embedding has, very-conservatively, approximately 13% greater page-walk overhead than ECC embedding.

5.3 Evaluation Results

Figure 5.11 shows the run time of each application with 4KiB pages, THP, and EMPTI broken down into the time spent in execution, the time spent servicing page walks, and overhead introduced by conflict and row-buffer embedding. We include both ideal results (with no conflict) and two results with 5% and 10% conflict. The results are normalized to the total run time with standard 4KiB pages. We also show the arithmetic mean across all run



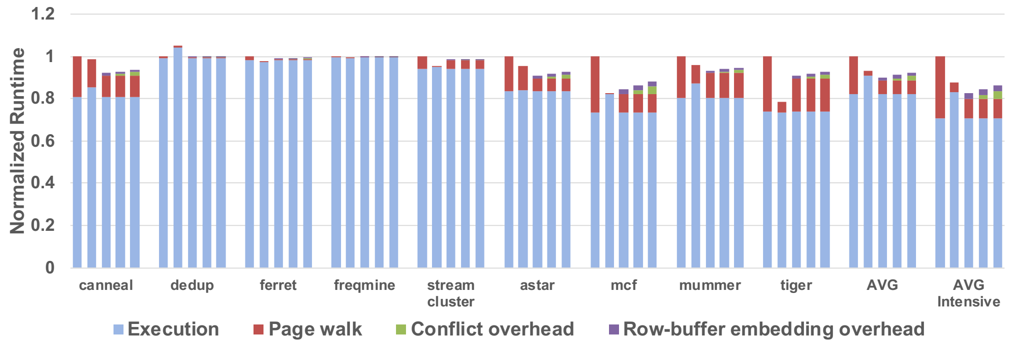


Figure 5.11: Performance of 4KiB pages, THP, Embedded Page Translation Information with 0%, 5%, 10% conflicts (left to right).

times and also the average across just those applications that are *page-walk intensive*; we define page-walk intensive applications as those applications in which more than 10% of the 4KiB execution time was spent servicing page walks (IS, Graph 500, GUPS, canneal, astar, mcf, mummer, and tiger).

There are three important takeaways from this experiment. First, EMPTI very effectively reduces the overhead of page walks. Across all applications, with EMPTI the page walk overhead (fraction of time spent on page walks) is 6.4% compared to 17.7% with 4KiB pages. The improvement is even more significant for page walk-intensive applications where EMPTI reduces the overhead from 29.36% to 9.27% on average and up to as high as a 6.5× improvement for GUPS.

Second, in all but two cases, EMPTI matches or exceeds the performance observed with THP. EMPTI reduces the overhead of fine-grained page walks to the point that they are either insignificant or are lower than the management overheads needed to maintain and create transparent huge pages in Linux. For example, both the CG.D and Graph500 benchmarks exhibit severe fragmentation with THP, which may lead to various management overheads. As a result, EMPTI performs significantly better with those applications. THP outperforms EMPTI by a negligible amount with the streamcluster benchmark and by about 11% with the tiger benchmark; tiger is the only benchmark where EMPTI is not advantageous. Overall, on average across all applications, THP and EMPTI improve performance over 4KiB pages by 6.6% and 11.2% on average, respectively. When looking only at page walk-intensive applications, the average improvement over 4KiB pages of THP and EMPTI are 12.2% and 20.1%, respectively.

Third, conflicts have limited impact. With a 5% conflict rate, on average, EMPTI still outperform 4KiB pages and THP by 18% and 6% in page walk-intensive applications.

Note that our results show much lower page-walk overhead with THP than prior research [46, 9]. This is because all our runs are without virtualization and because the Haswell processors we use have a much-improved TLB design compared to the older processors used in prior studies. The Haswell microarchitecture has a second level TLB that is both twice as large as earlier generations and which is shared between 4KiB and 2MiB pages.

Performance with Row-Buffer Embedding. With row-buffer embedding, EMPTI provides similar performance compared to the aggressive ECC embedding. This is because the additional overhead of accessing the EPTE is at most 13%, and in most cases far lower, depending on how heavily memory bandwidth is stressed—more latency-constrained applications are impacted less because the latency increase of back-to-back reads from the same DRAM row is small. If spatial locality is high, the overheads of row-buffer embedding are even lower.

Overall, EMPTI with row-buffer embedding still outperforms THP and 4KiB pages by 3.4% and 10% on average, respectively. For page walk-intensive applications, EMPTI reduces run time by 5.6% and 17.8% on average compared to THP and 4KiB pages, and even with a 5% conflict rate, it still reduces average run time by 3.4% and 15.6% respectively.

5.3.1 Energy Impact

We expect EMPTI to improve energy mostly because it reduces run time. EMPTI also reduces the number of memory accesses and row activations, but the impact through direct performance improvement is greater. When using ECC embedding, EMPTI trades off reliability for performance. If the same reliability tradeoff is made, but instead of embedding EPTEs, one of the redundant ECC devices in each rank is simply powered off, an equivalent-reliability baseline would have 5.6% lower DRAM energy. However, EMPTI saves more by reducing the total run time.

5.4 Future Work

As mentioned in Section 5.1, other coarse-grained virtual memory mechanisms, such as direct segments [9] and RMM [46], can be used for address mapping in EMPTI. In fact, EMPTI improves the flexibility of coarse-grained memory mappings. Because the fine-grained metadata is always maintained, exceptions within a region are allowed at the cost of falling back to the standard page-walk flow. Evaluation of these combined schemes is interesting, however, we leave them as future work.

Chapter 6

Efficient Metadata Caching

We discussed the inefficiency of current metadata caching schemes in Chapter 3 and in this Chapter we present the detailed design of Delta Caching—a novel metadata caching mechanism. By exploiting the redundancy of metadata, delta caching achieves up to 4× storage density compared to that in current state-of-the-art processors, without compromising the flexibility of address mapping and fine-grained metadata.

6.1 Motivation

As mentioned in Chapter 3, we observe that PTEs take substantial amount of capacity in the cache hierarchy and they are highly redundant. In this section, we analyse the detailed results. First, we evaluate the usage of data cache capacity for PTEs with a wide range of applications. Then, we evaluated the contiguity of memory mapping by comparing the number of deltas and pages needed to cover a certain percentage of memory accesses.

6.1.1 PTEs in the Data Cache

The processor cache hierarchy currently treats accesses to PTEs as data accesses—PTEs are cached.[1] PTEs and regular data compete for cache capacity, yet we are not aware of any details regarding any policies for this sharing. To characterize the cache use by PTEs, we conduct an experiment and estimate the effective LLC occupied by PTEs. We use the modified kernel from BadgerTrap [26] to intercept page walks, calling a custom fault handler for each walk. Within this handler we simulate an 8-way set associative cache with 64B cache lines for the accessed PTEs. We measure the hit rate of this cache for a range of cache capacities and compare these hit rates with a hardware performance counter that reports the rate of page walk accesses that miss the page hierarchy (in misses per million instructions, or MPMI). We use the simulated cache capacity that is closest to the hardware-measured MPMI to estimate the effective capacity of the LLC used for PTE caching. All results are measured/simulated on an Intel Core i5-4590 processor with 32GiB main memory.

We gather result from a set of memory intensive applications chosen from a wide variety of benchmark suites. Similar applications are also used in Chapter 5 We run only one single-threaded instance of each application from CPU 2006 and Bio Bench and set the thread count to 4 (one thread per core) for other multi-threaded applications. The detailed parameters of each

[1] In current Intel, AMD, and ARM processors, for example.

application are listed in Table 6.1.

Workload	Benchmark	Input	Effective cache capacity	Instruction count (Billions)
NAS parallel benchmark	IS	B	256KiB	11
		C	2048KiB	92
	CG	B	256KiB	228
		C	256KiB	614
	UA	B	64KiB	959
		C	256KiB	3804
SPEC CPU 2006	xalan	reference	512KiB	1149
	mcf		512KiB	325
	astar		256KiB	411
PARSEC	canneal	native	1024KiB	139
	streamcluster		128KiB	1060
Bio Bench	mummer	default	512KiB	12
	tiger		1024KiB	638
HPC challenge	GUPS	8GiB	4096KiB	330
		16GiB	4096KiB	654
Graph500	Graph500	scale 24	4096KiB	2992

Table 6.1: Benchmark parameters

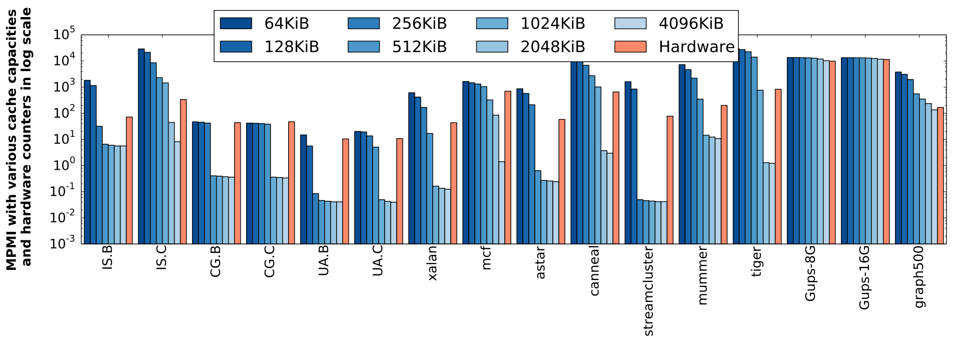


Figure 6.1: MPMI (misses per million instructions) of page walks with various cache capacities and hardware counters.

There are two important takeaways from the the result (Figure 6.1). First, the effective cache capacity for PTEs is highly application dependent. For example, CG uses only 64KiB of cache for PTE caching, however GUPS and graph500 likely use 4MiB (complete results are in Table 6.1). Second, increasing the effective cache capacity further reduces MPMI by a large amount in many applications, e.g., tiger achieves 1000x MPMI reduction by just doubling the effective cache capacity (from 1MiB to 2MiB).

6.1.2 Contiguity of Address Mapping

Previous work [89, 70] suggests that address mapping contiguity naturally exists beyond page boundaries, even with simple default behavior of current software. We study such (cross-page) contiguity using the applications in Table 6.1 running on a system with unmodified Linux (Kernel 3.12.13+). We use a Pintool [71] to measure the number of different pages touched and the number of different deltas (between the virtual and the physical addresses) required to cover different fractions of all memory accesses. For each application, we skip up to 100B instructions to bypass the initiation stage and collect the virtual and the physical addresses for every memory accesses for 10B instructions (or until the application completes) with the Linux pagemap interface [54].

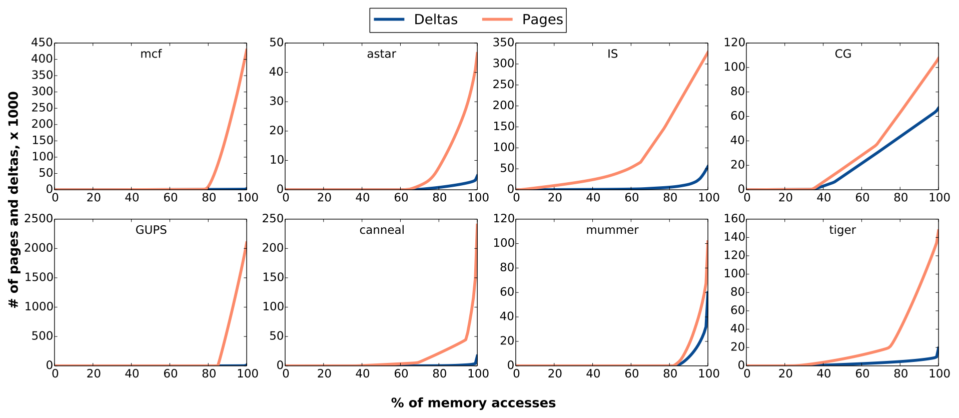


Figure 6.2: Number of pages and deltas over memory accesses.

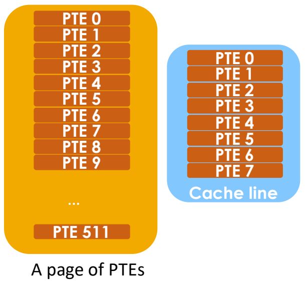
The results show two distinct types of behaviors (Figure 6.2). Some applications, e.g., mcf and GUPS, exhibit many fewer deltas than pages, indicating very strong cross-page contiguity. However, applications such as mummer and CG require a similar number of deltas and pages, indicating a highly randomized mapping.

Note that this contiguity is naturally occurring with an unmodified OS with all its default settings. No technique to enhance contiguity, such as a contiguous memory allocator in the Linux kernel or an eager allocator [47], are used. Furthermore, no single size is enforced, like it is with coarse-grained pages.

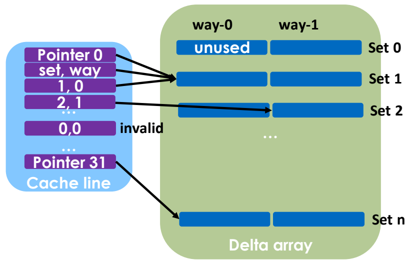
6.2 Delta Caching

We implement Delta Caching as part of the LLC, which is augmented with: (1) a cache-indexing function with which we can store delta-pointer PTEs within data cache lines and that is used to access such PTEs; (2) a mechanism to differentiate access to data or to a PTE and cache lines that store data or delta-pointer PTEs; (3) a cache partition that is used to store the delta and permission information and which is accessed by cache location rather than through a tagged cache lookup; and (4) a controller for managing the delta array.

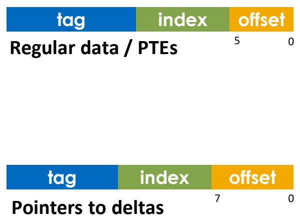
Before discussing these new features in full detail, we first describe one example access that uses Delta Caching: a TLB miss that queries the LLC for the corresponding PTE. A TLB miss is handled by the a hardware page walker, which marks the access as a *PTE access*. The PTE access uses a modified cache function to check whether a delta pointer exists for this PTE (Figure 6.3c). This function is designed for storing 32 delta pointers (representing up to 32 PTEs) within each 64B cache line. If the pointer tag matches (including verifying that the line indeed stores pointers using the PTE bit in the tag), the pointer is read from the cache line. A zero pointer indicates an invalid entry and the page walker proceeds to access main memory for the PTE. Otherwise, the pointer identifies a specific location within the delta array within the LLC (Figure 6.3b), which is used to construct the PTE and install it in the TLB. This access, along with the delta array management is depicted in Figure 6.4.



(a) Default PTE representation within a cache line.



(b) Delta-pointer based PTE representation.



(c) Cache address mapping for regular data/PTEs and pointers.

Figure 6.3: Delta Caching overview.

6.2.1 The Delta Array

While delta pointers are stored within the LLC and accessed with the modified cache function, we store deltas (really, (delta, metadata) pairs) in a separate storage array from the pointers. While this delta array can use a dedicated SRAM structure, we evaluate a design that embeds the delta array within the LLC, reducing the LLC capacity slightly. We store deltas in a set-associative array, which is indexed by the least significant bits (LSBs) of the delta. Each line contain one delta entry. We limit the total number of deltas to 2^{16}, such that each delta can be addressed with a 16-bit pointer. This strikes a balance between increasing PTE storage capacity in the LLC (allowing 32 delta-pointers per cache line) with a large number of possible delta entries to satisfy some applications (as suggested by the analysis of Section 6.1.2). The delta entry format is similar to a PTE, with the physical page number replaced by the delta in page number (Figure 6.5). The size of each delta entry remains 8B. In general, the needed capacity in the delta array may vary over time. However, for simplicity we only explore fixed-sized delta arrays, which we describe in Section 6.4.

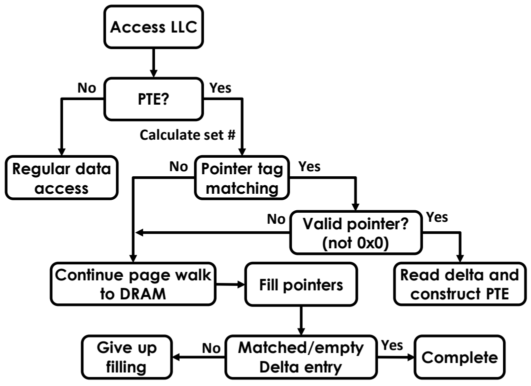


Figure 6.4: Flow of cache accessing

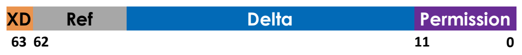


Figure 6.5: A delta array entry.

Delta Matching and PTE Fill. When a PTE access misses in its delta-pointer location, the PTEs read from main memory should be filled into the cache. Each PTE must match a delta entry in the delta array, or, attempt to install a new delta entry in the delta array. To do this a PTE's delta is computed and is compared with those already in the delta array. We use the least-significant bits of the delta value to index into the delta array and compare all delta values within that set of the delta array to the PTE delta. On a comparison match, the PTE is replaced with a pointer to the matched delta. Otherwise, if possible, one delta from the set is evicted and the pointer then updated. We discuss delta replacement below.

Delta Replacement. A stored delta cannot simply be evicted because there may be PTE pointers that still refer to it. One possible solution is to store back-pointers from deltas to PTE entries in the data cache, as done in [77]. However, the number of possible pointers in the large LLC, makes this approach prohibitively expensive. Instead, we propose to add a reference counter to the unused space in each delta entries metadata field (Figure 6.5). This counter is incremented every time a delta pointer that points to the delta entry is added to the LLC and is decremented when such a pointer is evicted from the LLC (as lines storing delta-pointer PTEs are evicted). We then use an LRU replacement policy on delta array, but only replace an entry with zero reference.

If no entry is found with zero references, the delta entry is not allocated and the PTE pointer that triggered the attempted delta replacement is set to zero to indicate a PTE cache access miss. In other words, it is possible for the cache line accessed for pointer-cached PTEs to miss.

6.2.2 Delta Caching Hierarchy

In our design, the delta array and pointer-based PTEs as stored in the LLC. However, it is possible the PTEs may also been cached closer to the cores. Thus, when the page walker accesses the cache hierarchy for a PTE, it starts at the L1 and L2 caches treating the PTE as data. Only at the LLC, the special cache function is used. This also implies that pointer-based PTEs are not propagated to other cache levels and are only used for TLB fills.

6.2.3 Delta Caching and Page Table Updates

Delta Caching stores PTEs in the cache using a different cache function than data. PTE accesses that are generated by the page walkers are easily identified and follow the flow discussed above. However, when the OS manipulates the page table, it uses regular store instructions to modify PTEs. Such PTE-stores also must use the Delta Caching cache index function to correctly update deltas, or at a minimum, invalidate delta-pointer PTEs in the cache (setting their delta pointer to zero). We propose to do this by introducing a new access mode (or new store instruction) that the OS will use for the purpose of modifying PTEs. This instruction is the only architectural modification required for Delta Caching and only the page-table manipulation routines of the OS need be aware of it.

When the processor executes such an instruction, the newly-written PTEs may be cached in the L1 or L2 data caches. However, the lines used to store PTEs are marked with a special PTE bit. This bit is used to correctly use pointer-based PTE storage when a cache line with PTEs is written back from the L2 to the LLC. On a writeback, the LLC controller checks this PTE bit. If it is set, the PTEs are used to fill pointer-based PTEs as already discussed. If the processor does not allow PTE caching in L1 and L2, the PTE-store invalidates the pointer of the PTE it modifies and directly stores the updated PTE to main memory. In this way, the next page walk will read the updated PTE from memory.

Note that the delta-pointer based PTEs do not include the accessed and modified bits of the actual PTE. The page walker directly updates those bits in main memory. This is acceptable because their update is rare and those bits are "sticky" and can be updated at any time by any core.

6.3 Dual PTE/Delta Caching

One issue with the basic Delta Caching introduced in previous section is that PTEs must either be cached as delta pointers or not cached in the LLC at all. As shown in Figure 6.2, some applications have a very large number of deltas and do not cache well as delta pointers. Such applications suffer when PTEs cannot be cached as data. To mitigate this problem, a simple idea is to allow PTEs to be stored in both delta caching mode and the regular direct data caching mode. To do this, a PTE access attempts two different LLC accesses at the two locations in which the PTE may be cached (as data or as a pointer). A simplistic implementation, however, will suffer from two deficiencies:

Extra Accesses — because of different address mappings, the two accesses mentioned above are very likely mapped to two different cache sets. Even for LLCs where tag array and data are accessed sequentially, only tag array is accessed twice. However, these two accesses require either longer latency (if issued sequentially), or more port contention on tag array (if issued simultaneously).

Capacity Inefficiency — if both forms of a PTE are cached frequently, the effective capacity will decrease. Checking for overlaps and flushing duplicated cache lines introduces complex and expensive mechanisms. For example, before adding a cache line of pointers, 4 corresponding cache lines (32 PTEs) need to be checked to detect and potentially flush all duplicated cache lines.

We propose a small, yet very effective change in cache address mapping to resolve both issues.

6.3.1 Address Mapping for Dual Caching

Instead of using the same cache address mapping as the regular data for PTEs, we shift the cache index by 2 bits such that PTEs and pointers use the same address bits for the cache index (Figure 6.6). This new mapping resolves the two issues above. First, PTEs and pointers for the same pages are always mapped to the same cache set, thus only one set in the tag array needs to be checked. Second, because the overlapped cache lines are placed in the same set, the cache replacement policy will automatically promote hit lines and demote lines that are not useful. When we hit both pointers and PTEs, we only promote the line with pointers. And when we miss both, a mode selection mechanism is used to make a decision.

With this mapping scheme, it is possible that one cache line with PTEs and another cache line with regular data have the same index and tag but form a different address. To resolve this conflict, we add one extra bit to the cache tag to indicate a cache line with PTEs/pointers. The PTE-store variant must also be modified to both invalidate a pointer and update a regular PTE, if both exist in the cache.

6.3.2 Mode Selection

The next question is how to dynamically decide whether to store PTEs as pointers or not. Ideally, delta caching start to be beneficial when the number of pointers in a cache line is greater than 8 (since there are 8 PTEs when storing directly). Instead of accurately calculating the number of valid pointers, which

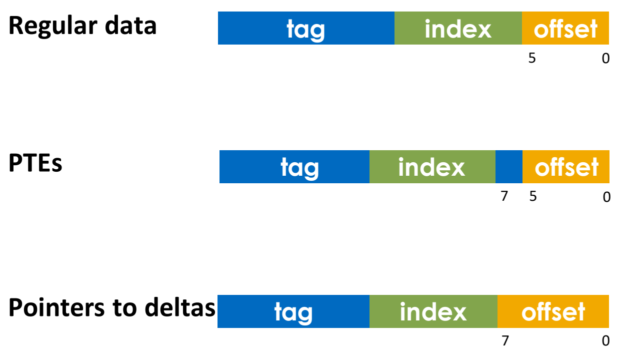


Figure 6.6: Address mapping for dual caching

requires validating all 32 pointers, we use a heuristic based on just the 8 PTEs fetched at one time from memory: if more than half of these PTEs can be assigned a delta entry, we allocate the line as for delta-pointer PTEs. Otherwise, we fall back to directly caching PTEs. While this scheme seems to be not accurate and only samples 8 out of 32 pointers, our evaluation shows that it is quite effective.

6.4 Evaluation Methodology

Our evaluation includes two parts. First, we check the effectiveness of the proposed delta caching mechanism by comparing the relative number of page walks miss LLC and require DRAM access with baseline (directly caching PTEs). Then, we use a performance model to evaluate the performance impact of delta caching.

Capacity	Baseline	Delta 50%		Delta 25%	
		Pointer	Delta	Pointer	Delta
64KiB	128*8	128*4	512*8	128*6	256*8
128KiB	256*8	256*4	1024*8	256*6	512*8
256KiB	512*8	512*4	2048*8	512*6	1024*8
512KiB	1024*8	1024*4	4096*8	1024*6	2048*8
1024KiB	2048*8	2048*4	8192*8	2048*6	4096*8
2048KiB	4096*8	4096*6	8192*8	4096*6	8192*8
4096KiB	8192*8	8192*7	8192*8	8192*7	8192*8

Table 6.2: Cache configurations, # of sets * # of ways

6.4.1 Page-Walk Memory Access Costs

To conduct a fair comparison, we use an equal share of LLC capacity for both schemes. The effective cache capacities used for PTEs have been profiled in Table 6.1. We use the same capacities for delta caching. For delta caching, this capacity needs to be partitioned to pointers and deltas. We include two sets of configurations for this partition. One attempts to split the total capacity $50-50$, and another attempts to assign 75% for pointers and 25% for deltas. This partitioning is done by removing a certain number of ways from the pointer array and allocating them to the delta array. To minimize the cost of delta array look-ups, we fix the associativity of the delta array to 8, such that only one 64B cache line is accessed for checking all deltas in a set. We also chose a maximum delta array of 2^{16} entries (512KiB), so that each pointer is limited to 16 bits and 32 pointers are stored in one 64B LLC line. After the size of delta array increased to maximum, we assign the remaining capacity to pointers. The detailed parameters for each size/configuration are shown in Table 6.2.

To collect the number of page-walk memory accesses for delta caching, we use the modified Linux kernel form BadgerTrap [26] to intercept page walks, and model both the baseline and delta caching scheme within the handler.

The goal of this evaluation is to verify that delta caching is able to reduce the number of page-walk memory accesses with the same capacity compared to baseline. However, it is also quite limited. First, it is difficult to dynamically adjust the size of the delta array with different applications. Second, a large fraction of page-walk memory accesses reduction does not mean substantial performance improvement. It is more interesting to check the impact on the run time. To address these concerns, we conduct the second part of the evaluation.

6.4.2 Comparing System Performance

As mentioned before, we use a fixed capacity for the delta array instead of adjusting the size for each application. This is a practical way to integrate delta caching into a processor. The delta array can either be added to the processor as extra storage or partitioned from the LLC. We evaluate three capacities of the delta array: 128KiB, 256KiB, and 512KiB.

We first use the same approach to evaluate the page-walk memory accesses. Now with the separately allocated delta array, we are expecting even more reduction, because the pointers can use the full capacity instead of the LLC sharing with deltas. Then we use an analytic model based on hardware performance counters measured on real hardware. The model is similar to the model used in Chapter 5. Note that delta caching does not just remove memory accesses for page walks, but rather, converts them into LLC accesses. We improve the model with separate page-walk overheads for cache and memory accesses. We first proportionally remove the time spent on page-walk memory accesses with the reduced page-walk LLC misses by delta caching. We then add a corresponding overhead in LLC accesses back to the total run time. The details of this model and parameters are summarized in Table 6.3.

Collected Statistics	
$T_{4K/THP}$	Total execution cycles with (4K / THP)
$PW_{4K/THP}$	Cycles spent in page-walks (4K / THP)
$PL_1/PL_2/PL_3/PL_M$	Page walks served by L1/L2/L3 cache and main memory (4K)

Parameters	
$C_1/C_2/C_3$	L1/L2/L3 cache latency (4/11/30 cycles)
R	Relative page walk memory accesses to baseline

Model	
Ideal execution cycles	$E_{4K/THP} = T_{4K/THP} - PW_{4K/THP}$
Page walk cache cycles	$PW_{Cache} = C_1 * PL_1 + C_2 * PL_2 + C_3 * PL_3$
Page walk memory cycles	$PW_{MEM} = PW_{4K} - PW_{Cache}$
Page walk cycles	$PW_{Delta\_cache} = PW_{Cache} + (1 - R) * PL_M * C_3$ $PW_{Delta\_mem} = R * PW_{MEM}$
Total cycles	$T_{Delta} = E_{4K} + PW_{Delta\_cache} + PW_{Delta\_mem}$

Table 6.3: Performance model and measured parameters. The 4K/THP notation indicates that two parameters are measured – once when running with 4KiB pages and a second time with THP.

6.5 Evaluation Results

We first show the normalized page-walk memory accesses for delta caching (Figure 6.7), and dual caching (Figure 6.8). Delta caching effectively reduces memory accesses from page walks on some applications with good cross-page contiguity. For instance, in IS, GUPS-8G, both configurations effectively reduce memory accesses generated by page walks to less than 50%. For canneal, the reduction to almost zero. However, as expected, applications with sub-optimal contiguity, e.g., CG and mummer, Delta Caching cannot help, and due to limitation of delta array size, DC actually increases page-walk LLC misses. On average (g-mean), delta caching reduces page-walk memory accesses to 69% with 50% of capacity as delta array, compared to baseline. The 25%-delta configuration, however, increase average page-walk memory accesses to 143%. We also show the average occupancy of pointer cache lines. Occupancy is defined as the number of valid pointers (with a valid delta entry) in a pointer cache line. We scan the whole cache every 10 million page walks to calculate the occupancy and report the average (arithmetic) result over the whole run. As we can see, IS and canneal achieves close to 32 average occupancy while CG suffers from low occupancy. For delta caching, most applications benefit from a larger delta array, and show a penalty when running out of delta entries (e.g., astar and stramcluster).

Dual caching achieves better, or at least similar, results than baseline (except for mummer). For applications that work well with delta caching, e.g., IS and canneal, the result is very close to delta caching. For applications that not work well with delta caching, e.g., CG and UA, dual caching significantly reduces the overhead introduced by delta caching. In addition, dual caching also achieves improvements with applications that run out of delta entries (astar and streamcluster in Figure 6.7). This is because the mode selection scheme detects the lack of delta entries and adapts to directly cache PTEs.

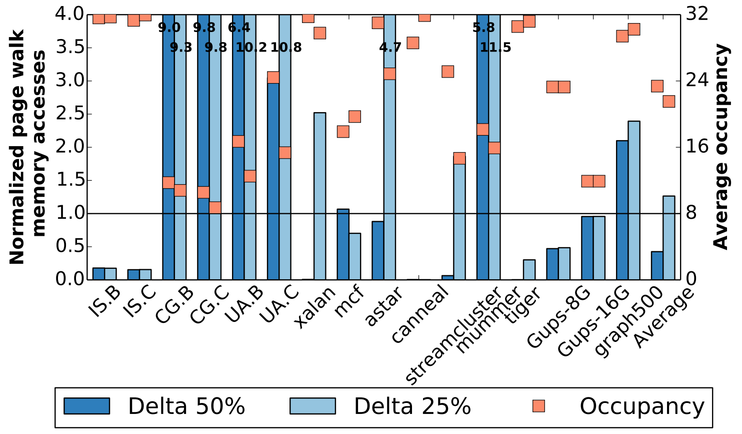


Figure 6.7: Normalized page walk-memory accesses with delta caching.

As mentioned in Section 6.4, we also evaluate the impact of dual caching with fixed capacities for the delta array. We compare the number of page-walk memory accesses of dual caching with the baseline. Differently from Figure 6.8, we now use a fixed capacity for the delta array. The result is similar to Figure 6.8, and because now we allocate delta arrays separately, and do not reduce the capacity for pointers, the overall result is better. With the smallest delta capacity, 128KiB, most applications outperform the baseline and achieve 89% page-walk memory access reduction on average (g-mean). For 256KiB and 512KiB delta arrays, the average reductions are even better (96% and 98%). For GUPS-8G, the reduction is almost to zero. This indicates that we achieve the maximum efficiency (one 2B pointer per 4KiB page, 4MiB of pointers for the 8GiB memory footprint of GUPS). The hit distribution shown in Figure 6.11 also explains this result. When 256KiB or 512KiB delta arrays are used, delta mode dominates the hits.

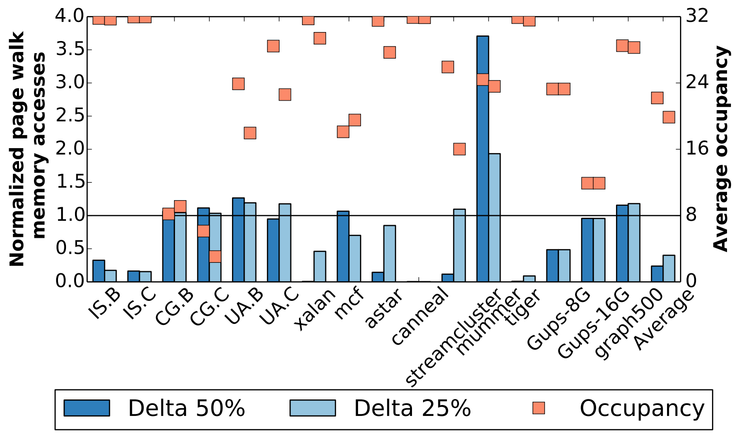


Figure 6.8: Normalized page walk-memory accesses with dual caching.

In Figure 6.12, we show the total run time distributed into execution time and page-walk servicing time. We compare the 4KiB pages baseline, THP (transparent huge page), and dual caching with 128KiB, 256KiB, and 512KiB delta arrays. Dual caching effectively reduces page-walk overhead in applications with good cross-page contiguity, e.g., IS, xalan, astar, GUPS-8G. On average, dual caching reduces page-walk overhead from 19.2% to (15.5%, 11.6% and 9.5% with 128KiB, 256KiB, 512KiB delta arrays). With the 512KiB configuration, dual caching reduces the overall run time by 10%, as good as THP. In some applications, dual caching yields sub-optimal improvement. For example in mcf, it only reduces page-walk overhead from 24% to 15% even with the largest 512KiB delta array. This is because dual caching only convert memory accesses in page walks into LLC accesses and some applications still spend substantial time on page-walk cache accesses. This can be improved by combining delta caching with other TLB coalescing techniques, e.g., CoLT [70].

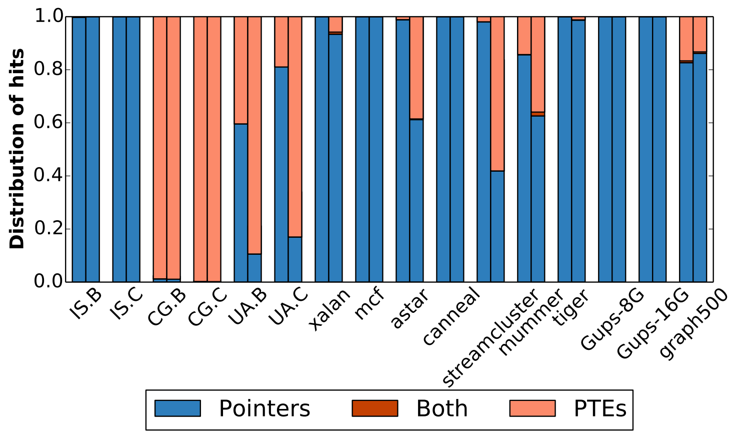


Figure 6.9: Distribution of hits with dual caching

We discuss this below.

6.6 Further Discussion

6.6.1 CoLT (Coalesced Large-Reach TLBs)

Delta caching utilizes mapping contiguity to reduce the number of memory accesses required by page-walks. Other techniques such as CoLT (Coalesced Large-Reach TLBs) [70] and Hybrid TLB Coalescing [64] focus on increasing the reach of TLBs to reduce the number of page walks.

In this section we compare delta caching with CoLT and also evaluate how CoLT and delta caching work together. We implement the set associative CoLT [70] as a second level TLB with 1K entries (128-set, 8-way each). Up to 8 PTEs can be coalesced and stored within each entry.

We model the performance impact of CoLT by comparing the number of page walks required after CoLT with the number of page walks without

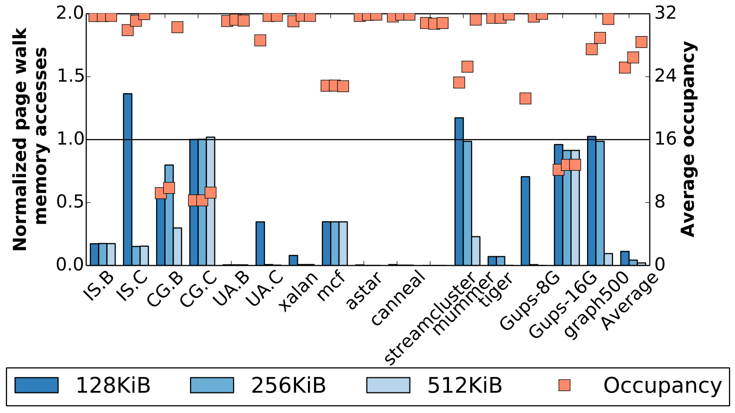


Figure 6.10: Normalized page walk memory accesses with fixed sized delta array

CoLT, and proportionally reduce the page walk overhead within cache. We observe that CoLT does not change the number of page walks miss the LLC. This is expected because the effective capacity of the cache used for PTEs is much larger than CoLT, although CoLT can be up to 8X more efficient than the regular TLBs. Finally, we evaluate how dual caching works together with CoLT. We apply dual caching to the LLC. Note that page-walks handled by dual caching are filtered by CoLT.

The results in Figure 6.13 show that when used alone, CoLT effectively improves TLB efficiency in some applications, e.g., mcf and astar, reducing the cycles spent on page-walk cache accesses from 8.2% and 6% to 3.2% and 1.2%. However, in applications in which memory accesses dominate the page-walk overhead, the improvement is very limited.

When combining CoLT and delta caching, page-walk overhead in both cache and memory is improved. The combined scheme achieves very small

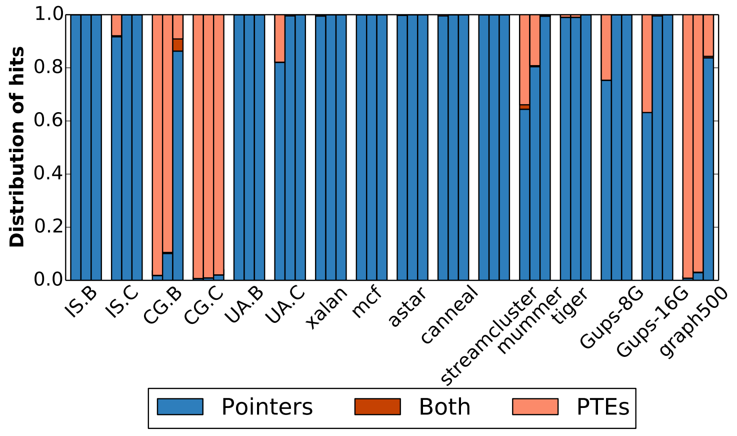


Figure 6.11: The distribution of hits (left to right: 128KiB, 256KiB, 512KiB)

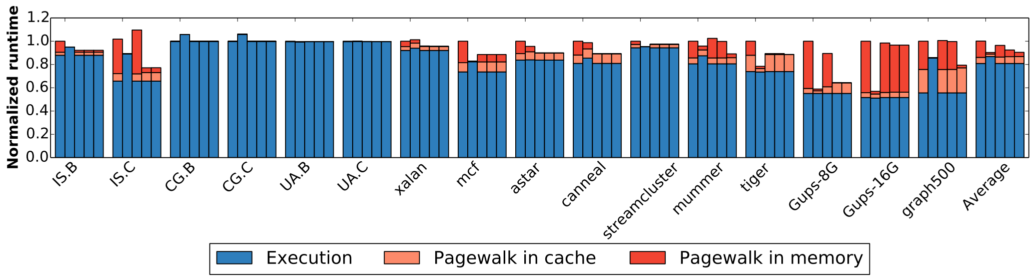


Figure 6.12: The distribution of run time (left to right: 4KiB Baseline, THP, Dual caching with 128KiB, 256KiB, 512KiB delta array)

page-walk overhead in IS.B and astar (only 1.3% and 2.3%) even with the smallest 128KiB delta array. And overall, the 512KiB configuration now reduces 59.7% of page walk overhead (19.2% to 7.9%), outperforming THP in total run time by 1.9% (88.5% v.s. 90.4%).

Because CoLT already exploits cross-page contiguity, the fraction of regular hits in dual caching increases. We show the distribution of hits in Figure 6.14. For example, compared to the hits distribution of using dual caching

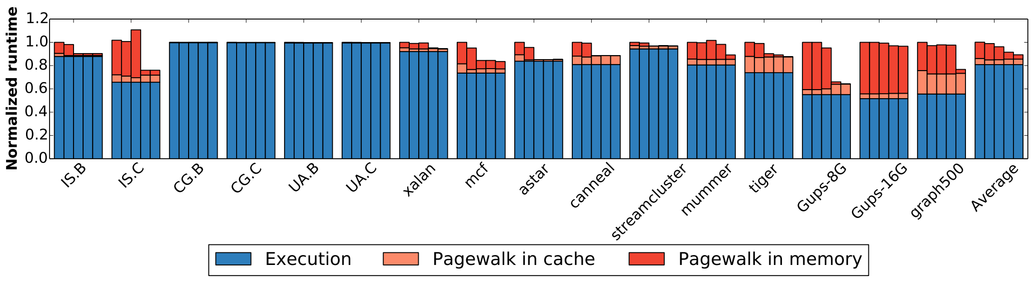


Figure 6.13: The distribution of run time (left to right: 4KiB Baseline, CoLT only, CoLT + Dual caching with 128KiB, 256KiB, 512KiB delta array

alone (Figure 6.11), the fraction of direct PTEs caching in xalan and tiger (128KiB) increases from almost zero to more than 10%. The mode selection scheme successfully detects the loss of contiguity and adapts to use direct PTEs caching.

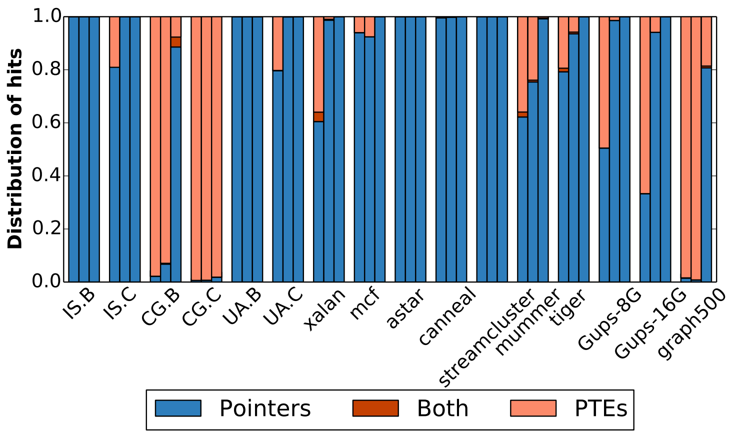


Figure 6.14: The distribution of hits (left to right: 128KiB, 256KiB, 512KiB)

6.6.2 Impact of LLC Capacity Reduction

We evaluate the impact of reserving part of the LLC capacity for the delta array on an Intel E5-2608L processor equipped with Cache Allocation Technology [38]. We allocate various number of ways in the LLC to control cache capacity from 8.25MiB to 6MiB. With Intel Performance Counter Monitors [39], we measure the IPC of each application with various LLC capacities. In addition to single/multi-threaded applications, we also evaluate the system with multi-programmed workloads, that place greater pressure on LLC resources.

Figure 6.15 shows normalized IPC over LLC capacities. Most workloads shows less than 3% IPC drop when LLC capacity is reduced from 8.25MiB to 7.5MiB. When LLC capacity is reduced to 6MiB, the performance impact becomes bigger, for example, mcf suffers a 9.7% IPC drop. However, since our delta array capacity is limited by 512KiB, the performance impact from reserving LLC capacity for the delta array is minimal.

6.6.3 Sensitivity Study to Memory Fragmentation

We use the memory fragmentation generator from ingens [52] to fragment 25% and 50% (a severe condition of operation) of memory, and compare the impact of delta caching with no artificial fragmentation (normal). We use dual caching with a 256KiB delta array to evaluate all three configurations.

Roughly half of applications show some impact from 25% fragmentation (Figure 6.16). On average, normalized page-walk memory accesses increase

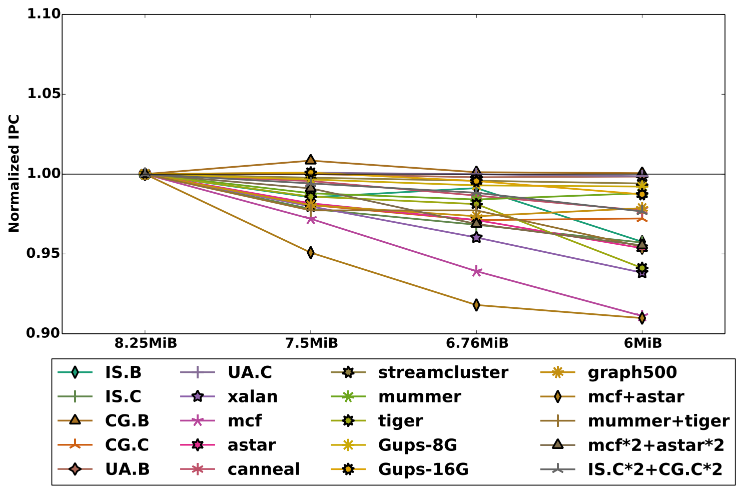


Figure 6.15: Impact of LLC capacity reduction

from 4% to 21%, but still much lower than the baseline. When fragmentation increases to 50%, most applications are impacted, with average memory accesses increasing to 53%.

6.7 Related Work

We discuss prior work relating to utilizing cross-page contiguity, include TLB coalescing to improve TLB efficiency [70, 64], and mechanisms that create or force the contiguity to coarsen virtual to physical mapping granularity [9, 47].

TLB Coalescing. TLB coalescing [70] improves the efficiency of the TLB and increases TLB coverage per entry. It detects instances of consecutive virtual to physical mappings and stores the mapping information with a single

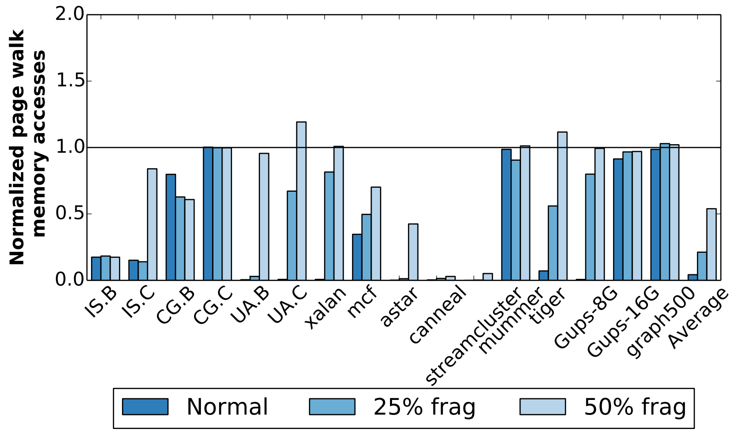


Figure 6.16: Normalized page walk-memory accesses with different fragmentation levels

TLB entry. The scope of coalescing is usually limited to 4-8 pages per entry to allow efficient lookups of coalesced TLB entries. While this restriction can be addressed with a fully-associative TLB, the number of fully-associative entries in TLB design is usually limited to 16-24 [40].

To enable a more-flexible coalescing granularity and better scalability, Park et al. [64] propose a HW-SW hybrid scheme where the OS can instruct HW with the optimal coalescing granularity (anchor size). The efficiency of this scheme relies on the anchor size selection algorithm. Any change of anchor size leads to invalidation of the entire TLB.

While TLB coalescing techniques effectively increase the coverage of the TLB, it is still limited by the number of TLB entries. Our results show that even for many applications with strong cross-page contiguity, more than 2000 regions/deltas/TLB entries are required to achieve 95% coverage, e.g., 2380 for astar, 23329 for IS, and 7759 for tiger, which is far beyond the size of TLBs (Figure 6.2).

Coarse-Grained Mapping. Superpages [82, 81, 63] and hugepages [14, 53, 5] enlarge the standard translation granularity to improve the efficiency of TLB and reduce the translation overhead. By enlarging the granularity of address mapping and management, the coverage of the TLB can be increased. However, applying a coarse-grained mapping is not free. Invoking hugepages explicitly usually requires porting and code modification and not transparent to applications. Relying on OS-managed transparent huge pages (THP) [5, 63] to form coarser pages adds overhead that can be significant[52]. This is apparent with the THP mechanism of Linux for which we measure and report significant management costs with some benchmarks. Larger page sizes increase the working set size [83, 52], rely on large contiguous memory regions, and fail to provide fine granularity protection. Holes between small non-continuous regions can lead to wasted memory. A single 4KiB dirty page can cause the write back of the whole mostly clean large page in current hardware implementations [5, 14]. Even for applications with suitable memory behavior, it is challenging to use coarse-grained pages because the optimal page size depends on the application, system, and dynamic characteristics of the inputs. Note that academic research on superpages addresses some of these limitations, but not all [82, 81]. For NUMA systems, large pages may lead to performance loss due to load imbalance and poor locality, which might entirely offset the benefits from fewer page walks [27].

Orthogonal to utilizing existing contiguity, schemes to create or force contiguity and to coarsen the granularity of mapping are also proposed [9, 47]. Direct segmentation [9] utilizes legacy segment mechanism to manage a one to one mapped region which is reserved in OS at boot time. To allocate into this special linear mapped region, the OS need to be argumented to guarantee and maintain that there is no holes in the entire reserved region.

Redundant Memory Mappings [47] increase the number of supported regions by adding range TLBs in parallel to the regular TLBs. A range-TLB entry can support an arbitrarily-sized region in which all pages are contiguously mapped. This contiguity is forced at allocation time by employing eager allocation.

The effectiveness of these contiguity-generating techniques relies on huge chunk allocations and hurt the flexibility of mapping, which is one of the most important benefits of virtual memory. Additionally, these region-based, TLB-like structures are usually limited by available resources. For example, Redundant Memory Mappings employ a 32-entry fully-associative range TLB. The precious entries need to be used carefully to maximize the benefits.

Memory Compression. Since PTEs are also data, general data compression techniques can be applied to PTEs as well. Previous work includes memory compression schemes for data in main memory [66, 2, 25, 50, 78], and caches [86, 65, 75, 3]. However, these compression and decompression algo-

	Application transparency	OS transparency	Size	Reach per entry	Loss of flexibility
Delta Caching	✓	mostly	Caches	32 pages per line	No
HW TLB coalescing [70]	✓	✓	TLBs	Up to 8 pages	No
Hybrid TLB coalescing [64]	✓	✗	TLBs	Up to 64K pages	Optional
Transparent Huge Pages [5, 63]	✓	✗	TLBs	2MiB	No
Redundant Memory Mappings [47]	✓	✗	TLBs	Unlimited	Yes
Direct segments [9]	✗	✗	1	Unlimited	Severe

Table 6.4: Comparison of delta caching with prior schemes for reducing translation overhead

rithms are substantially more complex than delta caching. Furthermore, the compression opportunities are usually exploited only within a certain scope (e.g., in the same cache line, PTE or in nearby blocks). However, delta caching utilizes the knowledge of PTEs format and address mapping contiguity. It is simple, yet especially efficient for PTEs.

We select Base-Delta-Immediate [65, 50] (BDI) as a representative compression scheme. The basic idea of BDI is representing data blocks with a low dynamic range as a base value and an array of differences. The combined size is much smaller than the original blocks. To store PTEs with the same delta, we can use one PTE as the base value and use 1 bit for each page to indicate if this page is valid (present and represented with the base value/PTE). This BDI-like scheme can be extended to multiple deltas/PTEs in the cache line with extra bits to indicate a specific delta. For example, 2 bits per page for 4 deltas. We implement a collection of BDI-like compression schemes to evaluate the efficiency of compression of PTEs and compare the normalized page-walk memory access count with delta caching and dual caching. We use X-Y to represent a compression scheme where in each 64B cache line, we store

X deltas and Y pages. Each page is represented with one of the deltas, or otherwise invalid.

Note that differently from previous evaluation, we use log scale to compare results because the dynamic range of the results from different schemes is very large. In Figure 6.17, we show normalized page-walk memory accesses for

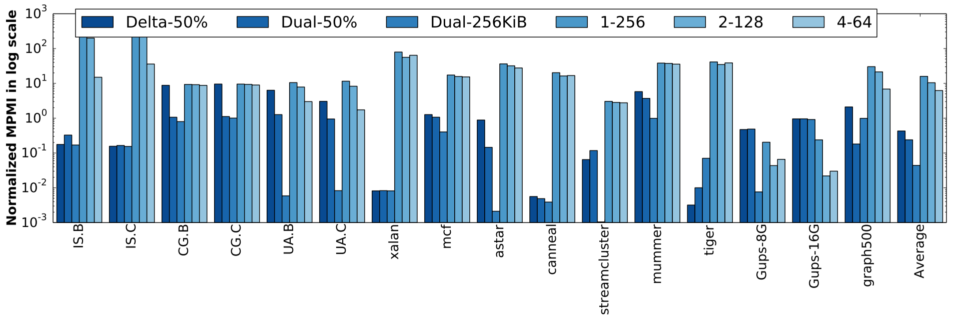


Figure 6.17: Normalized page walk memory accesses

delta caching, dual caching (all with 50% capacity for delta), and dual caching with 256KiB delta array, and compare these with three different compression-based schemes. Compression schemes are less effective than delta caching even though their maximum density (64 to 256 pages per line) is much larger than that of delta caching (32 pages per line). This is because compression schemes rely on contiguity within a relatively-larger range than delta caching. Pages can only be represented with one of the deltas within the same cache line. While pointers in delta caching can be mapped to any delta entry in the delta array. In addition, compression schemes may duplicate the same delta into multi cache lines, result in loss of efficiency. Compression performs well and outperform other schemes only with applications with extremely high contiguity, such as GUPS.

We summarize closely related schemes to reducing paging overheads and compare them with delta caching in Table 6.4. Delta caching is a mostly-transparent mechanism, makes no compromises on any existing benefits of fine-grained virtual memory, and only require minimal OS support (only changing the variant of a few existing store instructions).

Chapter 7

Conclusion

In this dissertation, I observe limitations in current fine-grained virtual memory: the long-latency of translation, the impact of page-grained address mapping, and the inefficient use of on-chip memory when caching metadata. I propose various mechanisms to avoid the unnecessary cost of fine-grained virtual memory. The proposed mechanisms approach the performance of coarse-grained virtual memory while maintaining seamless compatibility with fine-grained virtual memory and its benefits.

SIPT cache architecture enlarges the design space that constrained by fine-grained address mapping. It employ hardware mechanisms to guarantee correctness while leave opportunities for optional software optimizations. With proposed prediction mechanism, high speculation accuracy can be achieved with minimal cost.

EMPTI resolves the latency bottleneck of address translation. It breaks the limitation in pervious work that translation and metadata must be maintained in the same granularity. EMPTI decouples the translation and metadata, introduce novel schemes to embed metadata to near data places, effectively reduces the overhead of metadata accessing. With EMPTI, we achieves the advantage of both the coarse-grained translation and fine-grained metadata at the same time.

Delta Caching identifies the opportunities of more efficient metadata caching. It utilize the nature of contiguity in address mapping and the redundancy of metadata. With proposed novel caching scheme, delta caching achieves up to 4× density for storing PTEs while make zero compromise on the flexibility of address mapping. We also propose adaptive approach to avoid penalty on applications with sub-optimal contiguity. The adaptive version, dual caching achieve significant saving on page walk memory accesses and out performs THP when combined with TLB coalescing.

7.1 Future Work

Future virtual memory may need more variants of metadata with an even finer granularity to support emerging usages efficiently [59, 84]. The rest of this chapter outlines opportunities and challenges for future work.

Variants of Metadata. Current virtual memory supports per-page metadata within PTEs. This scheme is simple, yet both expensive and insufficient:

1. Existing metadata mechanisms maintain accurate states or counts. In many cases, the absolute accurate metadata is not required. For example, the per-page access count is used to identify and migrate hot pages in hetero-

geneous memory systems [59]. However, maintaining such an accurate counter in page granularity is expensive and unnecessary. A proper approximation for access count may be sufficient for performance optimization.

2. Many techniques, e.g., dynamic tainting [84] require even finer granularity than the 4KiB pages. Architectural support to store, manage and access these fine-grained metadata is an interesting research topic.

To conclude this dissertation. Current fine-grained paged virtual memory provides appealing features, but at significant cost. I propose and evaluate SIPT, EMPTI, and Delta Caching to address the overhead and constraints introduced by fine-grained virtual memory. These schemes identify the real implications of fine granularity, remove the unnecessary cost of fine-grained virtual memory, and achieve overhead comparable to coarse-grained virtual memory while maintaining all features of fine-grained virtual memory.

Appendices

Bibliography

[1] Amd64 architecture programmers manual, volume 2: System programming.

[2] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory expansion technology (mxt): Software support and performance. *IBM Journal of Research and Development*, 45(2):287–301, March 2001.

[3] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 212–223, June 2004.

[4] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ISPASS '05, pages 2–9, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Andrea Arcangeli. Transparent hugepage support. In *KVM Forum*, 2010.

[6] ARM Corp. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. 2014.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks&mdash;summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

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

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

[10] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 297–308, Washington, DC, USA, 2012. IEEE Computer Society.

[11] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Sup-*

port for Programming Languages and Operating Systems, ASPLOS XIII, pages 26–35, New York, NY, USA, 2008. ACM.

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

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

[14] Martin J Bligh and David Hansen. Linux memory management on larger machines. In *Proc. Linux Symposium*, 2003.

[15] Bruce Jacob. The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It.

[16] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst.*, 6(1):28–50, February 1988.

[17] Chiachen Chou, Aamer Jaleel, and Moinuddin K Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12. IEEE Computer Society, 2014.

[18] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965,*

Fall Joint Computer Conference, Part I, AFIPS '65 (Fall, part I), pages 185–196, New York, NY, USA, 1965. ACM.

[19] D. A. Patterson and J. L. Hennessy. Computer Organization and Design: The Hardware/Software Interface.

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

[21] R. Danilak. Transparent error correction code memory system and method, October 3 2006. US Patent 7,117,421.

[22] Intel Corporation Data Center Group. Intel xeon processor e7 family: Reliability, availability, and serviceability.

[23] Ronald G Dreslinski, Ali G Saidi, Trevor Mudge, and Steven K Reinhardt. Analysis of hardware prefetching across virtual page boundaries. In *Proceedings of the 4th international conference on Computing frontiers*, pages 13–22. ACM, 2007.

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

[25] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *Proceedings of the 32Nd Annual International Symposium on*

Computer Architecture, ISCA '05, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

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

[27] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[28] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[29] John F CouleurEdward L Glaser. Shared-access data processing system, 1968. US3412382A.

[30] James R. Goodman. Coherency for multiprocessor virtual address caches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS II, pages 72–81, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[31] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Ottawa Linux Symposium*, volume 1, pages 369–384. Citeseer, 2006.

[32] Graph 500. Graph 500. `http://www.graph500.org/`.

[33] M.J. Haertel, R.S. Polzin, A. Kocev, and M.B. Steinman. Ecc implementation in non-ecc components, March 13 2012. US Patent 8,135,935.

[34] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

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

[36] HP Labs. CACTI Pure RAM Interface. `http://quid.hpl.hp.com:9081/cacti/sram.y`.

[37] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ISLPED '99, pages 273–275, New York, NY, USA, 1999. ACM.

[38] Intel Corp. Cache monitoring technology and cache allocation technology.

[39] Intel Corp. Intel performance counter monitor - a better way to measure cpu utilization.

[40] Intel Corp. Tlbs, paging-structure caches, and their invalidation. 2008.

[41] Intel Corp. Intel® 64 and IA-32 Architectures Software Developer's Manual. 2016.

[42] Xun Jian, Henry Duwe, John Sartori, Vilas Sridharan, and Rakesh Kumar. Low-power, low-storage-overhead chipkill correct via multi-line error correction. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 24:1–24:12, New York, NY, USA, 2013. ACM.

[43] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 197–, Washington, DC, USA, 2001. IEEE Computer Society.

[44] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the distance for tlb prefetching: An application-driven study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 195–206, Washington, DC, USA, 2002. IEEE Computer Society.

[45] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift. Performance analysis of the memory management unit under scale-out

workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 1–12, Oct 2014.

[46] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 66–78. IEEE, 2015.

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

[48] Stefanos Kaxiras and Alberto Ros. A new perspective for efficient virtual-cache coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 535–546, New York, NY, USA, 2013. ACM.

[49] I. Kim and M. H. Lipasti. Understanding scheduling replay schemes. In *Software, IEE Proceedings-*, pages 198–209, Feb 2004.

[50] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. Bit-plane compression: Transforming data for better compression in many-

core architectures. In *the Proceedings of ISCA*, pages 329–340, Seoul, South Korea, June 2016.

[51] Jungrae Kim, Michael Sullivan, and Mattan Erez. Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory. In *the Proceedings of HPCA'15*, February 2015.

[52] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 705–721, Berkeley, CA, USA, 2016. USENIX Association.

[53] Linux Kernel Documentation. Huge TLB Page Support in Linux. `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`.

[54] Linux Kernel Documentation. Pagemap, from the userspace perspective. `https://www.kernel.org/doc/Documentation/vm/pagemap.txt`.

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

[56] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc

challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[57] Collin McCurdy, Alan L. Coxa, and Jeffrey Vetter. Investigating the tlb behavior of high-end scientific applications on commodity microprocessors. In *Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '08, pages 95–104, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Larry McVoy and Carl Staelin. LMbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[59] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, Feb 2015.

[60] Michael Kerrisk. Linux man-pages. `http://http://man7.org/linux/man-pages/man1/time.1.html`.

[61] Michael Kerrisk. Linux man-pages. `http://http://man7.org/linux/man-pages/man1/strace.1.html`.

[62] Muralimanohar, Naveen and Balasubramonian, Rajeev and Jouppi, Norman P. CACTI 6.0: A Tool to Model Large Caches. `www.hpl.hp.com/techreports/2009/HPL-2009-85.html`.

[63] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, December 2002.

[64] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 444–456, New York, NY, USA, 2017. ACM.

[65] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 377–388, New York, NY, USA, 2012. ACM.

[66] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 172–184, Dec 2013.

[67] A. Perais and A. Seznec. Practical data value speculation for future high-end processors. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 428–439, Feb 2014.

[68] Arthur Perais and André Seznec. Eole: Paving the way for an effective implementation of value prediction. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 481–492, Piscataway, NJ, USA, 2014. IEEE Press.

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

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

[71] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. Pin: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[72] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 469–480, New York, NY, USA, 2017. ACM.

[73] V. Salapura, J. R. Brunheroto, F. Redigolo, and A. Gara. Exploiting edram bandwidth with data prefetching: simulation and measurements. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 504–511, Oct 2007.

[74] Valentina Salapura, Robert Walkup, and Alan Gara. Exploiting workload parallelism for performance and power optimization in blue gene. *IEEE Micro*, 26(5):67–81, 2006.

[75] Somayeh Sardashti, Andre Seznec, and David A. Wood. Yet another compressed cache: A low-cost yet effective compressed cache. *ACM Trans. Archit. Code Optim.*, 13(3):27:1–27:25, September 2016.

[76] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based tlb preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 117–127, New York, NY, USA, 2000. ACM.

[77] A. Sembrant, E. Hagersten, and D. Black-Shaffer. Tlc: A tag-less cache for reducing dynamic first level cache energy. In *2013 46th Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 49–61, Dec 2013.

[78] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. Memzip: Exploring unconventional benefits from memory compression. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 638–649, Feb 2014.

[79] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM.

[80] spec.org. SPEC CPU 2017. `https://www.spec.org/cpu2017`.

[81] Mark Swanson, Leigh Stoller, and John Carter. Increasing tlb reach using superpages backed by shadow memory. In *ACM SIGARCH Computer Architecture News*, 1998.

[82] Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *ASPLOS*, 1994.

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

[84] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184, Feb 2008.

[85] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 273–284, Feb 2007.

[86] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 258–265, New York, NY, USA, 2000. ACM.

[87] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.

[88] Hongzhong Zheng, Jiang Lin, Zhao Zhang, E. Gorbatov, H. David, and Zhichun Zhu. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 210–221, Nov 2008.

[89] Tianhao Zheng, Haishan Zhu, and Mattan Erez. Sipt: Speculatively indexed, physically tagged caches. In *the Proceedings of the IEEE Interna-*

tional Symposium on High-Performance Computer Architecture (HPCA), pages 1–14, Vienna, Austria, February 2018.

Index

Vita

Tianhao Zheng was born in Xian, Shaanxi. He received the Bachelor of Computer Science and Technology from Tsinghua University in 2011. From August 2011, he started his Doctoral studies at The University of Texas at Austin. His research interest include different layers in memory sub-system and is specifically focused on virtual memory. He interned with Performance Architecture Team at Samsung Austin R&D Center (SARC) during the summer in 2014. He worked as a research intern with Architecture Research Team at Nvidia in Austin, Texas in 2015, and returned in 2016 as an intern in Compute Architecture Team in Santa Clara, California. He also interned with the Java Platform Team at Google in Sunnyvale, California in 2017.

Email address: thzheng@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.