ABSTRACT

SHIN, SEUNGHEE. Efficient Memory Architecture Design for Emerging Technologies. (Under the direction of Yan Solihin.)

While the amount of data needs to be processed is expected to increase exponentially in the near future, the advancement of system performance slows down due to physical limitations of transistor scaling. However, emerging memory technologies, die-stacked DRAM and Non-Volatile Main Memory (NVMM), are expected to become a new momentum for the future technology innovation by reducing performance overheads in memory accesses. Die-stacked DRAM technology enables a large Last Level Cache (LLC) that provides high bandwidth data access to the processor, and byte-addressable non-volatile memory technology allows programmers to store important data in data structures in memory instead of serializing it to the file system. On the other hand, GPUs have emerged as a first-class computing platform utilizing their massive parallel processing. In this advancement, shared virtual memory (SVM) across the CPU and the GPU is considered as one of the key features to promote GPUs into main processors by improving GPU programmability. However, we have identified difficulties in modern computer systems which are not yet prepared to efficiently utilize these new memory and GPU technologies. In this thesis, we introduce four novel approaches that propose new system architectures alleviating these difficulties.

Die-stacked DRAM is anticipated as huge LLC relieving bottlenecks in memory bandwidth, but it requires a large tag array that may take a significant portion of the on-chip SRAM budget. To reduce SRAM overhead, systems like Intel Haswell relies on a large block (Mblock) size. One drawback of a large Mblock size is that many bytes of an Mblock are not needed by the processor but are fetched into the cache. A recent technique (Footprint cache) to solve this problem works by dividing the Mblock into smaller blocks where only blocks predicted to be needed by the processor are brought into the LLC. While it helps to alleviate the excessive bandwidth consumption from fetching unneeded blocks, the capacity waste remains: only blocks that are predicted useful are fetched and allocated, and the remaining area of the Mblock is left empty, creating holes which is capacity overheads. In this thesis, we propose a new design, Dense Footprint Cache (DFC) which eliminates holes on top of Footprint cache by placing blocks in Mblock contiguously. Through simulation of Big Data applications, we show that DFC reduces LLC miss ratios by about 43%, speeds up applications by 9.5%, while consuming 4.3% less energy on average.

The NVMM is likely attached to the memory bus and allows processors to access them at word granularity in future systems. This can improve the system performance by eliminating the need of traversing file system every time persistent data is stored. However, modern systems reorder memory operations and utilize volatile caches for better performance, making it difficult to ensure a consistent state in NVMM. Intel recently announced a new set of persistence instructions, clflushopt,
clwb, and pcommit. These new instructions make it possible to implement fail-safe code on NVMM. In our experiments, we found that these persistence instructions in clusters along with expensive fence operations add a significant execution time overhead, on average by 20.3% over code with logging but without fence instructions to order persists. To deal with this overhead and alleviate the performance bottleneck, we propose to speculate past long latency persistency operations using checkpoint-based processing. Our speculative persistence architecture reduces the execution time overheads to only 3.6%.

Like the speculative persistence architecture, emerging non-volatile memory (NVM) technologies are encouraging the development of new architectures that support the challenges of persistent programming. An important remaining challenge is dealing with the high logging overheads introduced by durable transactions. In this thesis, we propose a new logging approach for durable transactions that achieves the favorable characteristics of both prior software and hardware approaches. We also propose a novel optimization at the memory controller that is enabled by a battery backed write pending queue in the memory controller. Since the WPQ is persistent, we drop log updates that have not yet written back to NVMM by the time a transaction is considered durable. We implemented our design on a cycle accurate simulator, MarssX86, and compared it against state-of-the-art hardware logging (ATOM [Jos17]) and a software only approach. Our experiments show that Proteus improves performance by 1.44-1.47×, on average, compared to a system without hardware logging and 9-11% faster than ATOM which also makes 3.4× more writes to memory than our design.

Recent studies on commercial hardware demonstrated that irregular GPU applications can bottleneck on virtual-to-physical address translations. In this work, we discover that the order of servicing a GPU’s address translation requests (specifically, page table walks) plays a key role in determining the amount of translation overhead experienced by an application. We find that different SIMD instructions executed by an application require vastly different amounts of work to service their address translation needs, primarily depending upon the number of distinct pages they access. We show that better forward progress is achieved by prioritizing translation requests from the instructions that require less work to service their address translation needs. Further, in the GPU’s Single-Instruction-Multiple-Thread (SIMT) execution paradigm, all threads that execute in lockstep (wavefront) need to finish operating on their respective data elements (and thus, finish their address translations) before the execution moves ahead. Thus, batching walk requests originating from the same SIMD instruction could reduce unnecessary stalls. We demonstrate that the reordering of translation requests based on the above principles improves the performance of several irregular GPU applications by 30% on average.
Efficient Memory Architecture Design for Emerging Technologies

by
Seunghee Shin

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Electrical and Computer Engineering

Raleigh, North Carolina
2018

APPROVED BY:

__________________________  ________________________
James Tuck                  Gregory Byrd

__________________________  ________________________
Ed Gehringer                Yan Solihin
Chair of Advisory Committee
DEDICATION

This dissertation is dedicated to my parents Youngsook Park and Youngsik Shin for their endless love and support.
Seunghee Shin is a Ph.D. student in Electrical and Computer Engineering department at North Carolina State University under Dr. Yan Solihin’s advising. His primary research interests lie in computer architecture and systems. Specifically, he has high interests in the impact of emerging technologies on memory systems. During his Ph.D. study, he has conducted research to explore better energy and performance efficient systems with die-stacked DRAMs and non-volatile memories (NVM). Additionally, during his internship in AMD research from 2017 to 2018, he has investigated the virtual address translation on GPUs. His studies have turned out publications at renowned venues such as International Symposium on Computer Architecture (ISCA) and IEEE/ACM International Symposium on Microarchitecture (MICRO). Besides, he has more than five years of professional system software development experiences. Especially, before his Ph.D. study, he worked in LG Electronics, Seoul, South Korea, from 2009 to 2012, where he engaged in multiple mobile system development projects, and he worked in iQstor Networks, Newbury Park, CA, where he worked on storage system developments from 2007 to 2009. He received MS degree in Computer Science from Northeastern University in 2007, where he studied computer networks, and BS degree in Computer Engineering from Myongji University, South Korea in 2005. He engaged in military service in South Korea from 2000 to 2002. He was born in Pohang, South Korea on December 9th, 1979 (officially, January 9th, 1980).
I would like to thank my advisor, Dr. Yan Solihin, for his help and encouragement during my Ph.D. study. He is an excellent role model as a good researcher to me. I learned what I needed to become a good researcher from him. His keen insight and professional knowledge always amazed me. I also appreciate his trust in me for such a long time and his financial support. My every experience at ARPERS is so valuable and will remain forever in my life.

I also would like to thank Dr. James Tuck. I highly enjoyed the collaboration with him during my Ph.D. study. I am glad that our works ended up being accepted to such reputable venues. I also appreciate his encouragement whenever I was struggling. His support always helped me to persevere all these years.

I would like to thank Dr. Greg Byrd and Dr. Edward Gehringher willing to become a member of my Ph.D. advisory committee. I especially appreciate Dr. Greg Byrd's recommendation letter for my job search. His support helped me to take the next step after Ph.D. degree successfully. I also thank Dr. Gehringher's help before my preliminary oral examination when I was in trouble to find a new committee member a few days before the exam date.

I would like to thank AMD research for offering me an internship opportunity in 2017. Especially, I want to thank my mentor, Dr. Arkaprava Basu, to help me to conduct GPU researches.

I am thankful to all my lab mates, especially, Yipeng Wang, Amro Awad, Satish Tirukkovalluri, Mohammad Alshboul, Nadim Jabbour, and Maryam Babaie. I would never forget the good time we had together and all the interesting discussions.

Finally, my Ph.D. would not have been possible without my parents, Youngsook Park and Youngsik Shin. My path was not always smooth. Whenever I feel frustrated, struggling, or sad, their support and trust lead me to persevere. Their endless love is the key for me to complete my degree. Thank you.
# TABLE OF CONTENTS

**LIST OF TABLES** ................................................................. viii  
**LIST OF FIGURES** ............................................................... ix  

**Chapter 1  INTRODUCTION** ......................................................... 1  
1.1 Stacked DRAM as a last level cache ............................................. 2  
1.2 Speculative Persistence on NVMM ............................................... 4  
1.3 Software supported hardware logging on NVMM ................................. 5  
1.4 Scheduling Page Table Walks for Irregular GPU Applications .................. 7  
1.5 Thesis layout ........................................................................... 9  

**Chapter 2  Dense Footprint Cache: Capacity-Efficient Die-Stacked DRAM Last Level Cache** 10  
2.1 Related Work ............................................................................ 10  
2.2 Overview ................................................................................... 12  
2.3 Dense Footprint Cache Design ...................................................... 15  
2.3.1 Cache Replacement Policy ..................................................... 15  
2.3.2 Placement Fit Policy ............................................................. 19  
2.3.3 Dealing with Block Miss ......................................................... 20  
2.3.4 Early Footprint Update .......................................................... 21  
2.3.5 Overhead analysis ................................................................ 22  
2.4 Methodology ............................................................................. 23  
2.5 Evaluation .................................................................................. 25  
2.5.1 Overall Performance .............................................................. 25  
2.5.2 Replacement Policy Comparison ............................................. 28  
2.5.3 Placement Policy ................................................................. 30  
2.5.4 Early Footprint Update .......................................................... 30  
2.6 Sensitivity Study ....................................................................... 31  

**Chapter 3  Hiding the Long Latency of Persist Barriers Using Speculative Execution** .... 33  
3.1 Background .............................................................................. 33  
3.1.1 Memory Persistency Models ................................................... 33  
3.1.2 Intel PMEM Persistency Model ............................................... 34  
3.2 Workloads ............................................................................... 36  
3.2.1 Failure Safety through Transactions Using Write-Ahead Logging ....... 37  
3.2.2 Workload Construction ........................................................ 39  
3.3 Speculative Persistence ............................................................... 42  
3.3.1 Architectural Requirements .................................................... 42  
3.3.2 Implementation .................................................................... 44  
3.4 Methodology ............................................................................. 48  
3.4.1 Simulation configuration ......................................................... 48  
3.4.2 Benchmarks ......................................................................... 49  
3.5 Evaluation ................................................................................. 49  
3.5.1 Overall Performance ............................................................. 49
LIST OF TABLES

Table 2.1  The baseline system configuration. ................................. 24
Table 2.2  L2 cache size configurations as affected by a limited SRAM budget. .... 25
Table 2.3  L3 miss ratios of Footprint cache (FC) vs. our DFC. All numbers are percentages. 27
Table 2.4  Predictor accuracy comparison of Footprint cache vs DFC. All numbers are percentages. ................................................................. 28

Table 3.1  Benchmarks constructed for our study. For all benchmarks, we size each node to be 64 bytes and align them to cache blocks. Thus, to persist one node update, one clwb will be required. ................................................................. 39
Table 3.2  The baseline system configuration. ........................................ 48
Table 3.3  SSB configurations and parameters. ......................................... 49

Table 4.1  The baseline system configuration. ........................................ 68
Table 4.2  Benchmarks constructed for our study. Except for SS which has 256 bytes for each string, we size each node to be 64 bytes and align them to cache blocks in all benchmarks. Thus, to persist one node update, one clwb will be required. InitOps and SimOps are expressed in terms of the number of operations that are executed per thread. ................................................................. 69
Table 4.3  Speedups for large transactions. ........................................... 75
Table 4.4  LLT miss rate (%) for different benchmarks. ............................. 76

Table 5.1  The baseline system configuration. ........................................ 89
Table 5.2  GPU benchmarks for our study. ........................................... 90
LIST OF FIGURES

Figure 1.1 Logging model taxonomy .................................................. 6
Figure 2.1 Illustration of the Footprint cache [Jev13] (a), and Dense Footprint cache (b). 13
Figure 2.2 Comparing three possible DFC replacement policies: LRU+, MaxAvg, and MaxMin. ................................................................. 16
Figure 2.3 Block miss handling in DFC. ............................................. 21
Figure 2.4 Speedup ratios of Footprint cache (Footprint), Unison (Unison), our design (DFC), and small-block cache (64B block), for when the L3 cache size is 64MB (a) or 256MB (b), relative to system with no stacked DRAM L3 cache. Full Stacked DRAM is an oracle L3 cache with zero miss rate and zero tag overheads. 26
Figure 2.5 Comparing LRU+, MaxAvg, and MaxMin replacement policies for a 64MB DFC in terms of miss ratios (a) and speedups over Footprint cache (b). 29
Figure 2.6 Miss-ratio Comparison for Early Footprint Update ............ 30
Figure 2.7 Performance Comparison over number of tags difference per set 31
Figure 2.8 Performance Comparison under memory contention .......... 32
Figure 3.1 Intel persistence instructions .......................................... 35
Figure 3.2 Persistence example with linked list ................................. 38
Figure 3.3 Inconsistent data structure example with linked list .......... 39
Figure 3.4 Rebalancing with incremental logging ........................... 41
Figure 3.5 Rebalancing with full logging ....................................... 42
Figure 3.6 Speculative persistence design ...................................... 45
Figure 3.7 Speculative persistence ................................................. 47
Figure 3.8 The successive additional overheads from logging (Log), PMEM instructions (Log + P), sfence instructions (Log+P+Sf), versus our scheme (SP256), normalized to code without logging and persistency. ................................................................. 50
Figure 3.9 The ratio of instruction count to baseline instruction count. 51
Figure 3.10 The ratio of fetch queue stall cycles to baseline execution cycles. 51
Figure 3.11 The maximum number of in-flight pcommits .................. 52
Figure 3.12 The average number of speculative stores while a pcommit is outstanding. 52
Figure 3.13 The overhead of different configurations for speculative store buffer size over baseline. ......................................................... 53
Figure 3.14 The rates of bloom filter's false positive. .......................... 53
Figure 4.1 Steps needed to implement fail-safe undo logging in software. 58
Figure 4.2 Comparison between software logging and hardware logging. 59
Figure 4.3 Example code transformation for our SSHL approach. ........ 61
Figure 4.4 Proteus hardware design .................................................. 64
Figure 4.5 Speedup comparison on NVMM, with software logging with PMEM as baseline. 70
Figure 4.6 The normalized stall cycles of the pipeline front-end ........ 71
Figure 4.7 The number of NVMM writes, normalized to PMEM with no logging. 72
Figure 4.8 Speedup comparison on slow NVMM (300ns for write), with SW logging as baseline(PMEM). 73
| Figure 4.9  | Speedup comparison on DRAM, with software logging with PMEM as baseline. | 73 |
| Figure 4.10 | Speedup comparison with varying LogQ sizes. The baseline is software logging with PMEM. | 74 |
| Figure 4.11 | Speedup comparison with varying LPQ and LogQ sizes. The baseline is software logging with PMEM. | 75 |
| Figure 5.1  | Baseline system architecture. | 79 |
| Figure 5.2  | Performance impact of page walk scheduling. | 81 |
| Figure 5.3  | Distribution of number of memory accesses (i.e., 'work') for servicing address translation needs of SIMD instructions. | 82 |
| Figure 5.4  | Impact of interleaving of a GPU's page walk requests. | 83 |
| Figure 5.5  | Fraction of instructions whose page walk requests are interleaved with requests from other instructions. | 84 |
| Figure 5.6  | Average latencies of the first- and the last-completed page walk request per instruction. | 84 |
| Figure 5.7  | Key components and actions of the SIMT-aware page table walk scheduler. | 85 |
| Figure 5.8  | Speedup with SIMT-aware page walk scheduler. | 91 |
| Figure 5.9  | GPU stall cycles in execution stage. | 91 |
| Figure 5.10 | Latency gap between the first and the last-completed page walk request per instruction. | 92 |
| Figure 5.11 | Number of page walk requests with SIMT-aware scheduler normalized over FCFS. | 92 |
| Figure 5.12 | Number of active wavefronts accessing the GPU's L2 TLB with SIMT-aware scheduler (normalized over FCFS). | 93 |
| Figure 5.13 | Speedups with varying GPU L2 TLB size and page table walker counts. | 94 |
| Figure 5.14 | Speedups with varying IOMMU buffer size. | 95 |
We are facing the end of Moore's law and Dennard scaling [Sim16]. Since transistor scaling is slowing down, the continued demand for power and performance efficient systems forces computer architects to re-think traditional computer architectures and search for alternative ways to realize greater computing power. Interestingly, this change has driven computer architects to analyze software and design specialized hardware that accelerate crucial software algorithms rather than merely waiting for the next generation of silicon technologies and software to take advantage of them. In this trend, the community pays attention to new device technologies such as die-stacked DRAM and non-volatile memory (NVM). The die-stacked DRAM can provide much higher bandwidth than traditional DRAM, and the non-volatile memory can provide higher density than DRAM and non-volatility at latencies competitive with the DRAM. GPUs also attract the computer architecture community by their massive parallelism. However, current GPUs are only designed as the secondary processor to provide extra computing power to the CPU by copying data from CPU to GPU and then back again. As an effort to elevate the GPU as a primary computing processor, computer architects introduced the heterogeneous system architecture (HSA) [Kyr12] which tightly integrates the CPU and GPU in a processor and provides a unified main memory space. However, we are also facing new architectural challenges from these new technologies, because modern systems have been optimized for conventional CPUs and memories for decades. These challenges require systems to develop new designs in order to maximize the benefits from these emerging technologies.

This thesis faces those architectural challenges. Specifically, it discusses the impact of adopting
new technologies on the memory system. The memory hierarchy is a critical part of computer architecture. The memory wall problem caused by the gap between the CPU speed and the memory speed has been around for a long time. In addition, the advances in chip fabrication have enabled more and more cores to be integrated on a single chip. The increase in the number of cores is increasing the demand for memory bandwidth. Hence, lack of sufficient memory bandwidth can become a crucial performance limiting factor [Rog09]. To overcome these limitations, modern systems have adopted multiple memory layers composed of different memory technologies. However, memory systems encounter new challenges. Some of the non-volatile memories are competing with DRAM for use as future main memory. These new non-volatile main memory (NVMM) technologies are changing the traditional memory hierarchy enormously by removing a long-time held distinction between memory and storage. Memory manufacturers are also facing the DRAM scaling limitation. However, the integration of the CPU and GPU increases the memory bandwidth demands significantly. In the following sections, we introduce the difficulties in these new technologies and discuss our solutions for each problem.

1.1 Stacked DRAM as a last level cache

Last decade, the advance in chip fabrication has enabled more and more cores to be integrated on a single chip. The increase in the number of cores is placing a significant pressure on the memory bandwidth demand, creating a bandwidth wall problem where memory bandwidth increasingly becomes a performance limiting factor [Rog09]. Die-stacked DRAM has alleviated this bottleneck by providing high bandwidth memory near the processor. A stacked DRAM may be used as a Last Level Cache (LLC) [Jev13; Jia10; Loh09; LH11; QL12; Zha07; Jev14; Cho15; Lee15; Gul14; OL15], as a part of the main memory [Mes15; Don10], or both at the same time. As part of the main memory, a stacked DRAM incurs no tag overheads but requires careful page allocation and migration to keep hot pages in the stacked DRAM. In addition to the need of hardware support for profiling, frequent migrations are intolerably expensive; e.g., the costs of TLB shootdown [Mes15]. In contrast, as an LLC, a stacked DRAM is automatically managed by hardware but incurs tag overheads.

One of main challenges of using a stacked DRAM as the LLC is that it requires a large tag array. If the tag array is implemented in SRAM, tag access will be very fast, but it consumes a large portion of the SRAM budget. For example, for a 256MB 16-way associative DRAM LLC operating on 64-bit addresses, the tag array requires a 32MB SRAM budget, which is prohibitively expensive. There are two general approaches in dealing with this problem. In one approach, tags are co-located in the stacked DRAM together with data [LH11; QL12; Jev14; Gul14]. While it completely solves the tag array overhead problem, this approach comes at a very significant cost in latency and bandwidth. A typical cache access requires three (rather than one) DRAM accesses: the first one to read out the tag from the stacked DRAM, the second access to read out data upon a cache hit, and the third one
to update the state and LRU bits. Recognizing the problem, recent proposals try to mitigate this overhead by using way predictions or traffic filter [Jev14; Gul14; Cho15], but they reduce bandwidth overheads at the expense of lower hit rates or still suffers high miss latency while improving hit latency. Thus, in [Jev14], the authors report performance improvement only when the LLC size is very large (1GB or larger).

The second approach to solve the tag array overhead problem is to rely on large block sizes. A large block (major block or Mblock) in the range of a few kilobytes significantly reduces the number of cache blocks and thus the tag array size [Jia10; Jev13; Jev14]. For example, if the block size is 2KB instead of 64B, the 256 MB LLC only requires 1MB in SRAM budget for the tag array. This approach is used in the Intel Haswell system [Ham14]. The main drawback of this approach is cache capacity and bandwidth waste: insufficient spatial locality in most programs result in only a few small blocks (i.e, 64B blocks) being accessed out of a 2KB Mblock that is fetched and allocated in the LLC. There are several strategies to mitigate the waste. In CHOP [Jia10], a "hot page" filter prevents cold Mblocks from getting allocated in the LLC. Another strategy is to subdivide the Mblock into smaller blocks where only blocks predicted to be needed by the processor are brought into the LLC [Jev13; Jev14]. Thus, an Mblock may not be fully filled: only blocks that are predicted useful are fetched and allocated, and the remaining area of the Mblock are left empty (holes). Such a strategy removes bandwidth waste but not the cache capacity waste.

Holes are costly with stacked DRAM for two reasons. First, holes incur capacity overheads which could have been used for useful data. In addition, it incurs power overheads because DRAM rows containing holes still need to be refreshed, even when no valid data is present in holes. To eliminate the drawback, in chapter 2, we propose a new design that we refer to as Dense Footprint cache (DFC), similar to [Sol14]. Like Footprint cache, DFC uses a large Mblock and relies on useful block prediction in order to reduce memory bandwidth consumption. However, when blocks of an Mblock are fetched, they are placed contiguously in the cache, thereby eliminating holes. By eliminating holes, DFC avoids wasted LLC capacity and hosts more valid blocks in the stacked DRAM. However, due to eliminating holes, the Mblocks in DFC have variable sizes and a cache set has a variable associativity. This introduces unique challenge to the cache placement and replacement policy.

A large incoming Mblock may require multiple consecutive resident Mblocks to be evicted. Traditional replacement policies such as LRU are not applicable in this case because they assume there is only one victim Mblock. Thus, we propose three new cache replacement policies for DFC: LRU+, MaxAvg, and MaxMin. LRU+ starts with LRU Mblock and expands to adjacent Mblocks. MaxAvg and MaxMin calculate the average and minimum stack distance positions of all candidate Mblock groups and select a group with the maximum score. We analyze these replacement policies and derive their behavior bounds mathematically and examine their performance through simulations. In addition, we also examine augmenting the replacement policy with placement policies such as best fit and worst fit.
Another contribution is a new footprint prediction update policy. In Footprint cache, a footprint of an Mblock is updated when the Mblock is evicted. However, when the LLC is large, Mblocks stay too long in the cache and hence the footprint predictor learns very slowly. We solve this problem by allowing footprints to be updated even when Mblocks are still resident in the cache. We refer to this as *Early Footprint Update* policy.

Finally, we evaluate DFC using a set of big data applications, and found that it reduces LLC miss ratio by 43% compared to Footprint cache, which translates into an average speedup of 9.5%.

### 1.2 Speculative Persistence on NVMM

While avoiding the file system can potentially provide a large performance advantage, it introduces a major challenge of ensuring the consistency of data across unexpected hardware or software failures. In a conventional system, the order in which store values are written back to the main memory from the last level cache (LLC) does not follow the program order, so a failure produces an unpredictable outcome of which stores have their values permanently reflected (i.e. *durable*) in the NVMM. Thus, to achieve *failure safety*, two critical components are required: (1) a specification or model of *persists* (i.e. when stores will be durable in the NVMM), and (2) rewriting of software so that it can recover safely upon a failure based on the persistency model.

Various persistency models have been proposed [Pel14; Con09; Jos15; Int16d; Lu14], including the use of transactions [Kol16b; Vol11]. Our starting point is Intel PMEM [Int16d], a persistency model supported by several new instructions, such as *clwb*, *clflushopt*, and *pcommit*, used in conjunction with existing x86 instructions such as *clflush* and *sfence*. *Clflushopt* and *clwb* force dirty data out of the cache hierarchy, while *pcommit* acts as a persist barrier by forcing a flush of the write buffers in the memory controller. These instructions may be re-ordered by the processor with respect to non-dependent loads and stores, hence fence instructions are needed to precisely control when data is made durable at the NVMM. While these instructions have been announced, their performance impact has not been studied or characterized in the context of NVM workloads, and few workloads have been written using these new instructions. Similarly, ARM has recently introduced a new instruction, DC CVAP, for persistence support in ARMv8.2 [ARM16].

To gain a deeper understanding of the programming and performance challenges of persistent data structures on future processors and applications, we rewrite several data structures and kernels commonly found in databases and file systems, to incorporate failure safety using the PMEM persistency model. Like prior work [Kol16b; Vol11], we adopt transactional semantics using write-ahead logging as the basis for reasoning about failure safety. Building and using a microarchitectural simulator extended to support PMEM, we make several observations. First, write-ahead logging requires frequent use of *sfence-pcommit-sfence* instruction sequences, to correctly order persists in these benchmarks. Second, the PMEM instructions occur in clusters with fence operations, have
long latencies, and incur a significant performance penalty, causing execution time overheads of up to 55%, and 21% on average, on top of logging overheads. The performance penalty arises primarily due to the pipeline stalling for the completion of the `sfence-pcommit-sfence` instruction sequence.

Based on these observations, we propose **speculative persistence** (SP), architectural support to speculatively execute past long latency persist barriers to hide their latency and reduce their impact on performance. Speculative execution is triggered when a persist barrier stalls the pipeline. The analysis of our benchmarks shows that such barriers can take 100s to 1000s of cycles to complete. Rather than waiting, in SP, a checkpoint is taken at the persist barrier, the sfence is speculatively retired, and the processor proceeds speculatively retiring the sfence and following instructions. Meanwhile, the pcommit completes non-speculatively in the background. Stores are buffered and not allowed to propagate to memory until the pcommit finishes and speculation completes. Buffering of speculative state and conflict detection proceed in much the same way as prior speculation schemes.

In the past, speculative execution has been proposed for other purposes such as hiding L2 miss latency [Cez06; Mut03; Kir05; Sri04], improving memory consistency model performance, and speculating past synchronization operations [MT02; RG01; HM93]. While bearing some similarities with prior speculation techniques, SP faces new challenges. First, other persist barriers are likely to occur in the shadow of the current speculative persist barrier. Because these instructions may need to flush data out of the cache, they cannot execute as part of the speculative region and must instead be buffered and played back at commit of the speculative region. Second, because some instructions must be delayed and played back later, this limits how far we can speculate. To overcome these challenges, we design multiple checkpoints to speculate across multiple persist barriers. The speculative regions commit sequentially as each pending persist barrier completes in sequence, thereby ensuring proper transactional semantics.

We evaluate our new architecture and compared it against the same system without speculation. Our experiments show that SP reduces the execution time overheads to only 4.5% on average, compared to code with logging but without PMEM instructions.

### 1.3 Software supported hardware logging on NVMM

As we have introduced in earlier section, one persistency model that is easier for programmers to achieve failure safety is that of **durable transactions** [Kol16b; Vol11; Int16d]. With a durable transaction, all stores in a transaction persist or none of them do.\(^1\) This is a simple and useful abstraction for programmers.

\(^1\)Note a fundamental difference between durable transaction and transactional memory (TM): a durable transaction specifies when data is made durable in NVMM, whereas TM deals with when data is visible to other threads. Consequently, durable transactions apply even to sequential code.
In chapter 4, we explore a key challenge of using durable transactions: how to perform logging efficiently and flexibly. Durable transactions require logging, either through *redo* or *undo* logging. The log allows the transaction to be recovered if a failure occurs during the transaction. A log can be created through software code inserted by the programmer, through a library [Int16d], or directly in hardware without additional code [Jos17]. Software approaches (SW) incur large performance overheads due to additional instructions but offer the greatest flexibility, including unlimited transaction size and control over logging operations. The latter approach (HW) [Jos17] has low performance overheads, but it is typically less flexible.

In this paper, we propose a new logging approach, referred to as *Proteus*, that achieves the favorable characteristics of both software and hardware approaches: software controls its own log space, it can support unlimited transactions of arbitrary size, it can manage its own recovery, and it has low overhead. Our approach introduces two new instructions that indicate whether a load instruction should create a log entry and a log flush instruction to write the log entry to NVM. We presume no additional programming effort beyond specifying transaction boundaries, since the compiler can generate instructions appropriately for code inside transactions. Additional hardware support is introduced, largely within the core, to manage the execution of these instructions and critical ordering requirements between logging operations and updates to data to ensure durable transaction semantics. *Proteus* avoids any limitation on the size or number of transactions through judicious design of the interface: software remains in control of allocating the log space, and hardware keeps the cost of updating the log low. The taxonomy in Table 1.1 illustrates each logging model’s pros and cons.

We also consider integrating *Proteus* with a battery backed WPQ, allowing the WPQ to be considered part of the persistency domain. Once writes reach the WPQ they are considered durable.
The presence of a battery-backed WPQ is consequential: it presents a new opportunity to avoid writes to the NVMM. A key observation that we exploit is that most logs are created and discarded, because failures are rare. Thus, we apply an optimization where we distinguish data blocks in the WPQ that are there for logging or not. This distinction allows us to treat them differently, where log blocks are kept as long as possible in the WPQ and discarded when a transaction commits, whereas non-log blocks are allowed to drain from the WPQ to the NVMM. Not only does this optimization improve performance, but more importantly, it helps in extending the lifespan of NVMM by avoiding many writes to the NVMM.

We implemented Proteus on a cycle accurate simulator, MarssX86, and compared it against state-of-the-art hardware logging (ATOM [Jos17]) and a software only approach. Our experiments show that Proteus improves performance by 1.48×, on average, compared to a system without hardware logging and 10.5% faster than ATOM. A significant advantage of our approach is dropping writes to the log when they are not needed. On average, ATOM makes 2.11× more writes to memory. Even though stores are often not on the critical path, persistent writes are critical given the store-ordering constraints required for durable transactions.

1.4 Scheduling Page Table Walks for Irregular GPU Applications

GPUs have emerged as a first-class computing platform. The massive data parallelism of GPUs had first been leveraged by highly-structured parallel tasks such as matrix multiplications. However, GPUs have more recently found use across a broader range of application such as graph analytics, deep learning, weather modeling, data analytics, computer-aided-design, oil and gas exploration, medical imaging, and computational finance [NVI16]. Memory accesses from many of these emerging applications demonstrate a larger degree of irregularity – accesses are less structured and are often data dependent. Consequently, they show low spatial locality [Cha14b; Bur12; Men10].

Irregular memory accesses can be particularly harmful to the GPU’s Single-Instruction-Multi-Threaded (SIMT) execution paradigm where typically 32 to 64 threads (also called workitems) execute in a lockstep fashion (referred to as wavefronts or warps) [Men10; Ves16; SK12; Rog13; Tar09; Wan15a; Wan15]. When a wavefront issues a SIMD memory instruction (e.g., load/store), the instruction cannot complete until data for all workitems in the wavefront are available. This is not a problem for well-structured parallel programs with regular memory access patterns where workitems in a wavefront typically access cache lines from only one or a few unique pages. The GPU hardware exploits this to gain efficiency by coalescing multiple accesses into a single access. For irregular applications, however, memory accesses of workitems within a wavefront executing the same SIMD memory instruction can access different cache lines from different pages. This leaves little scope for coalescing and leads to memory access divergence – i.e., execution of a single SIMD instruction could require multiple cache accesses (when accesses fall on distinct cache lines) [Rog13;
Tar09; Wan15a; Wan15] and multiple virtual-to-physical address translations (when accesses fall on distinct pages) [Ves16; Pic14].

A recent study on real hardware demonstrated that such divergent memory accesses can slow down an irregular GPU application by up to $3.7\times-4\times$ due to address translation overheads alone [Ves16]. The study found that the negative impact of divergence could be greater on address translation than on the caches. Compared to one memory access on a cache miss, a miss in the TLB\(^2\) triggers a page table walk that could take up to four sequential memory accesses in the prevalent x86-64 or ARM architectures. Further, cache accesses cannot start until the corresponding address translation completes as modern GPUs tend to employ physical caches.

In this work, we explore ways to reduce address translation overheads of irregular GPU applications. While previous studies in this domain primarily focused on the design of TLBs, page table walkers, and page walk caches [LP14; Aus18; Pic14], we show that the order in which page table walk requests are serviced is also critical. We demonstrate that better scheduling of page table walks can speed up applications by 30% over a baseline first-come-first-serve (FCFS) approach. In contrast, naive random scheduling can slow applications down by 26%, underscoring the need of a good schedule for page table walks.

We observe that page walk scheduling is particularly important for a GPU’s SIMT execution. An irregular application with divergent memory accesses can generate multiple uncoalesced address translation requests while executing a single SIMD memory instruction. For a typical 32-64 wide wavefront, execution of a single SIMD memory instruction by a wavefront can generate between 1 to 32 or 64 address translation requests. Due to the lack of sufficient spatial locality in such irregular applications, these requests often miss in TLBs, each generating a page table walk request. Furthermore, servicing a page table walk requires anything between one to four sequential memory accesses. Consequently, servicing address translation needs of a single SIMD memory instruction can require between 0 to 256 memory accesses. In the presence of such a wide variance in the amount of work (quantified by the number of memory accesses) required to complete address translation for an instruction, we propose a SIMT-aware page walk scheduler that prioritizes walk requests from instructions that would require less work. This aids forward progress by allowing wavefronts with less address translation traffic to complete faster.

Further, page walk requests generated by a single SIMD instruction often get interleaved with requests from other concurrently executing instructions. Interleaving occurs as multiple independent streams of requests percolate through a shared TLB hierarchy. However, in a GPU’s SIMT execution model, it does not help a SIMD instruction to make progress if only a subset of its page walk requests is serviced. Therefore, servicing page walk requests in a simple first-come-first-serve (FCFS) order can impede the progress of wavefronts. Our proposed scheduler thus also batches requests from

---

\(^2\)Translation Lookaside Buffer or TLB is a cache of address translation entries. A hit in the TLB is fast, but a miss triggers long-latency page table walk to locate the desired address translation from an in-memory page table.
the same SIMD instruction for them to be serviced temporally together. The SIMT-aware scheduler speeds up a set of irregular GPU applications by 30%, on average, over FCFS.

To summarize, we make two key contributions:

- We demonstrate that the order of servicing page table walks significantly impacts the address translation overhead experienced by irregular GPU applications.

- We then propose a SIMT-aware page table walk scheduler that speeds up applications by up to 41%.

1.5 Thesis layout

The rest of this thesis is laid out as follows. Chapter 2 discusses the detail of DFC and simulation results. Chapter 3 proposes the speculative persistence and experiment results. Chapter 4 proposes Proteus. Chapter 5 proposes page table walk scheduler. Finally, Chapter 6 concludes this thesis.
In Chapter 2 we propose our new last level cache design using stacked DRAM. This chapter is organized as follows. Section 2.1 discusses related work, Section 2.2 overviews the DFC, Section 2.3 presents DFC design details, and Section 2.4 describes the evaluation methodology. At last, Section 2.5 and 2.6 discuss evaluation results and key findings.

2.1 Related Work

Researchers have recognized the usefulness of stacked DRAM for a long time. Several earlier studies demonstrated the high bandwidth benefit of using stacked DRAM as the entire main memory [Woo10; Kgi06; Liu05; Loh08]. More recent studies recognize that off-chip DRAM will exist alongside stacked DRAM, hence they have proposed to use the stack DRAM as a part of the main memory [Mes15; Don10]. Concurrently, other studies explored the design and usefulness of using the stacked DRAM as a hardware managed last level cache (LLC) [Jev13; Jia10; Loh09; LH11; QL12; Zha07; Jev14; Cho15; Lee15; Gul14].

In the context of using stacked DRAM as the LLC, there was a recognition that the SRAM budget
used for the tag array of the LLC is substantial and thus needs to be reduced [Jia10]. In one approach, tags are co-located in the stacked DRAM together with data (LHcache [LH11], Alloy cache [QL12], Unison cache [Jev14], and Bi-modal cache [Gul14]). As discussed earlier, while it completely solves the tag array overhead problem, this approach increases latency and bandwidth significantly, as a typical cache access requires three (rather than one) DRAM accesses: one to read out the tag, another to read out data, and another to update the state and LRU bits. To reduce such a high hit latency, Unison and Bi-modal cache rely on way predictions [Jev14; Gul14]. However, a cache miss (or way prediction miss) still require multiple times DRAM accesses, and when coupled with high cache miss rates in LLC, the performance overhead is still considerable [Jev14]. In [Jev14], the authors reported that Footprint cache as showing better performance than Unison cache when the DRAM LLC, unless the LLC is very large (1GB or larger). There is also a proposal to reduce bandwidth overheads in these cache designs at the expense of lower hit rates [Cho15].

The second approach is to rely on large block sizes, such as the 1KB Mblock size in the Intel Haswell 128MB L4 cache [Ham14]. The main drawback of using a large Mblock is bandwidth consumption due to insufficient spatial locality in most programs, where only a fraction of the bytes in an Mblock is actually used by the processor. To mitigate that, CHOP [Jia10] uses a “hot page” filter to avoid allocating cold Mblocks from being allocated in the LLC. Another solution, used in Footprint cache [Jev13] and Unison cache [Jev14], subdivides the Mblock into blocks and only brings blocks that are predicted useful into the LLC [Jev13]. Blocks that are not brought in are left empty (holes). Footprint cache [Jev13] is the closest related work to this paper. In contrast to Footprint cache, our proposal Dense footprint cache (DFC) eliminates holes by storing valid blocks contiguously. This improves the usage of cache capacity. As a result, although virtually an Mblock size is 2KB, its physical size in the stacked DRAM LLC is variable. In a way, DFC is a compressed form of Footprint cache. With the same DRAM budget, DFC achieves the performance comparable to twice the size of Footprint cache.

Our approach DFC’s co-location of multiple Mblocks in a cache line resembles decoupled sectored cache [Sez94]’s co-location of multiple blocks, applied to a much larger granularity. However, apart from the granularity, there is a key difference. In the decoupled sectored cache, blocks are not stored in a de-fragmented format, they are simply co-located. Thus, two blocks can be co-located only when their non-hole sub-blocks do not overlap. In DFC, Mblocks are actually stored in a de-fragmented format, hence it is not important for blocks in one Mblock to strictly correspond to a hole in another Mblock for them to be co-located. This ensures higher co-location success. Finally, a similar idea to DFC can be found in a patent [Sol14], however the patent only provides organization of the cache but not its management, and does not provide quantitative evaluation.
2.2 Overview

As discussed earlier, the tag array overhead is a major cost of stacked DRAM LLC. One important requirement for the tag array is that it must be accessible at a low latency in order to facilitate quick determination of LLC hit or miss. This is one of the primary reasons why the tag array of the LLC is implemented in SRAM in the Intel Haswell system \([\text{Ham14}]\). In most stacked DRAM products, such as the Hybrid Memory Cube (HMC), the stacked DRAM access latency is only slightly faster than the off-chip DRAM access latency, primarily because design choices were made to prioritize density over fast access time in both cases. A solution is to increase the cache block size so that it reduces the number of blocks to keep track and makes SRAM tag array viable. There is a limit to this approach, however. If the LLC is very huge, e.g. 1GB or larger, the required SRAM budget for the tag array is going to be too large. However, we believe that diminishing return from cache capacity will likely cap the LLC capacity much below 1GB, leaving the remainder of the stacked DRAM capacity as a part of the main memory to avoid duplicating not useful data in cache and main memory.

In the Footprint cache, an Mblock size of several KBs is used (e.g. 2KB). Each 2KB Mblock is divided into $32 \times 64$-byte blocks. Figure 2.1(a) illustrates a Footprint cache. The cache has a SRAM tag array and DRAM data array. The tag array shows which blocks in an Mblock are valid (represented by ‘1’s), and which ones are holes (represented by ‘0’s). Correspondingly, the data array shows valid blocks as solid-colored cells and holes as crossed-out cells. The figure shows 4 Mblocks: A, B, C, and D.

Suppose that a request comes from the processor or L1/L2/L3 cache (Step 1). After checking the tag array, suppose that the block is found on Mblock A, as indicated by the valid bit vector. Each bit in the valid bit vector represents whether the block is valid or is a hole in the Mblock. In this example, the block is then returned to the processor (Step 2a). Let us consider a different situation, where the access is to Mblock E which cannot be found in the Footprint LLC. Here we have an Mblock miss, the Footprint History Table (FHT) is looked up. The Footprint in the FHT indicates that in the past, only the first two blocks of the Mblock are accessed by the processor. Hence, only these two blocks will be fetched from the main memory (Step 3) and allocated in the LLC. An Mblock that is evicted to make room for the new Mblock updates the FHT, where the valid bits are recorded as the footprint for the evicted Mblock (Step 4). Note that as an overview, the discussion is somewhat simplified here. The actual Footprint cache uses two bits (valid and dirty) per Mblock to indicate whether an Mblock is valid or not, clean or not, and has been demanded or not. Also, the FHT is indexed using the hash of the PC of the missing load/store, and the address being missed. The Footprint cache may also be set associative.

Figure 2.1(b) illustrates our proposal: *Dense footprint cache* (DFC). The DFC in the figure co-locates up to two Mblocks in the same cache line (or DRAM row). The figure shows the tag array now consists of more Mblocks per cache line. Each Mblock not only has the tag and valid bit vector,
but also has a starting location that indicates the offset in the cache line where the physical location of the Mblock starts. In the figure, Mblock A and B are co-located in one cache line. Mblock C and D overflow the second cache line, hence only one Mblock (i.e. Mblock D) can be kept in the cache. Note that the tag array of DFC has the same size as in Footprint cache, however the data array is half the size of Footprint cache. The total number of Mblocks that can be placed in the DFC falls somewhere between the size of Footprint cache and twice its size, depending on the average number of valid bits of Mblocks.

As shown in the figure, DFC may still contain holes. However, there are two key differences between holes in DFC versus ones in Footprint cache. In Footprint cache, holes arise due to \textit{internal fragmentation} within an Mblock, where some blocks are not valid within an Mblock. With DFC, there are no holes due to internal fragmentation; holes are caused by \textit{external fragmentation}, where
co-located Mblocks do not occupy the entire cache line. Note also that the holes in Footprint cache are *permanent*: holes in an Mblock stay for the entire residency of the Mblock in the cache, and disappear only when the Mblock is evicted. In contrast, holes in DFC are *temporary*, and can disappear when a new Mblock that can utilize the space is allocated in the cache. For example, in the figure, if a new Mblock with a size of two blocks is brought into the second cache set, the holes will disappear. Because of these two major differences in nature of holes in Footprint cache vs. DFC, there are much fewer holes in DFC, resulting in better space efficiency (fewer cache capacity misses) and better power efficiency (refreshes are applied mostly to valid data).

While the cache becomes denser, there are several unique challenges for designing a DFC. Unlike the Footprint or conventional caches, the physical size of Mblocks is different because the number of blocks that are allocated in the DFC differs across Mblocks. This introduces several new challenges to the cache placement and replacement policies. Let us consider a case where an incoming Mblock needs to be allocated in the cache. Suppose evicting an existing Mblock creates sufficient space but there are several possible Mblocks with different sizes that can be chosen. In this case, the cache replacement policy faces a similar problem to the one faced by memory management in an operating system (OS), where the requested memory size may not be equal to the sizes of blocks in the free space. Several options, such as *best fit*, *worst fit*, *first fit*, or *random fit* are possible. In addition to such options, a question arises of how to incorporate traditional replacement policy information, such as stack position, when choosing which Mblock to evict.

Let us also consider a case where multiple Mblocks need to be evicted to make sufficient space for the incoming Mblock. Now we not only face the fit and stack position considerations, but we also face a plurality of stack positions to consider. For example, we may be able to evict a group of two Mblocks that have stack positions three and eight, versus a group of two Mblocks that have stack positions five and six. The first group of Mblocks contain the least recently used Mblock (stack position 8) but also a more recently used Mblock. The second group of Mblocks contain two of some of the least recently used blocks. A new placement and replacement policy is needed to choose which group of Mblocks to evict.

A final problem that we deal with DFC is an observation of a problem that we discover with the FHT update policy. The FHT is updated only when an Mblock is evicted. The blocks that are actually accessed during the Mblock’s residency in the LLC form a new footprint in the FHT entry for the Mblock. Such a policy makes sense in that when there is a miss to the Mblock in the future, the footprint provides good prediction of what blocks to fetch and allocate in the LLC. However, we found that as the LLC size grows, Mblocks stay longer in the cache and are evicted less frequently, which reduces footprint updates. This slows down the ability for the FHT to learn footprints. To overcome this slow footprint learning, we propose a new footprint update technique called *early footprint update* (EFU), which allows footprints to be updated even when the Mblock still resides in the cache. We showed that EFU improves footprint learning and improves the footprint prediction.
coverage. The following section will discuss these design issues in more detail.

2.3 Dense Footprint Cache Design

In DFC, Mblocks have different actual sizes, so we need a new cache placement/replacement policy. There are two types of cache misses in DFC: an Mblock miss where the access is to an Mblock that is not found in the cache, and a block miss where the access is to a block that is not found in an Mblock that is currently cached. An Mblock miss arises due to cold/capacity/conflict, while a block miss arises from inaccurate prediction of the footprint of an Mblock such that the Mblock was allocated in the cache without having the needed block. We will discuss the Mblock miss first.

While all Mblocks share the same virtual size (e.g. 2KB), their actual sizes vary and depend on how many blocks are allocated in the cache. When a large incoming Mblock needs to be allocated, we may need to evict several Mblocks to make room for the incoming Mblock. The traditional least recently used (LRU) replacement policy is not directly applicable when dealing with multiple victim Mblocks. A relevant question is: how should we take into account stack recency positions when evicting multiple Mblocks? Another question is: how should fragmentation be taken into account in the eviction? And finally: how should we handle block miss for an Mblock that is already in the cache?

2.3.1 Cache Replacement Policy

As mentioned earlier, potentially several Mblocks need to be evicted to provide space for an incoming Mblock. The first constraint that we choose is to require the evicted Mblocks to be consecutive, in order to avoid shifting the remaining Mblocks to create contiguous space large enough for the incoming Mblock. Shifting is very expensive in terms of latency and power consumption and hence we would like to avoid it.

An alternative to shifting is to split the incoming Mblock into non-contiguous parts so that it fits in the non-contiguous space freed up by evicting non-contiguous victim Mblocks. Breaking an Mblock rather than allocating it contiguously in a cache line creates unnecessary complexities in the design, as the tag array has to be designed to record multiple starting addresses. Reading from an Mblock requires assembling from non-contiguous parts, and additional computation to locate the desired block in an Mblock. Thus, we also avoid this.

Given our design choice's preference of evicting contiguous Mblocks, we can now narrow down the replacement policy by considering the stack positions of the victim Mblocks. Here we consider three different policies: LRU+, MaxAvg, and MaxMin. LRU+ straightforward: it first selects the LRU Mblock as a victim. If that does not free up sufficient space, its neighboring Mblock (either to the left or to the right) is selected for eviction as well. Which neighboring Mblock is selected? LRU+ selects
the less recently used Mblock between the two neighboring Mblocks to add to the LRU Mblock. If these two Mblocks still have not freed sufficient space, we repeat the process by expanding the victim by another neighboring Mblock until the total eviction size becomes equal or larger than the requested block size. Figure 2.2 (top diagram) illustrates LRU+ replacement policy. The figure shows the tag array of a cache line, showing there are 8 Mblocks currently cached in the line: A–H. Each Mblock’s LRU stack position is shown with a number indicating MRU (0), 2nd MRU (1), and so on until LRU (7). The LRU Mblock is D. Other fields, such as the starting location of an Mblock, its state, and which blocks are present, are omitted in the figure.

Now suppose that there is a miss to a new Mblock that is twice as large as any of the existing Mblocks. Thus, two Mblocks need to be evicted. With LRU+, we expand the victim group from Mblock D to its neighboring Mblock C or Mblock E. Since Mblock E is closer to the LRU (stack position 2), the two Mblocks selected for eviction are Mblocks D and E. Note, however, Mblock E is the third MRU block, hence it is risky to evict it since the processor may still need it. To reduce the risk, we propose two additional policies: MaxAvg and MaxMin.

In the MaxAvg, groups of Mblocks are scored based on their average stack distance positions, which is shown in the numbers above the Mblocks. For example, a group of two Mblocks A and B.
have an average stack distances of $\frac{4+6}{2} = 5$, a group of two Mblocks B and C have an average stack distances of $\frac{6+1}{2} = 3.5$, etc. After these average scores are calculated, the group of Mblocks having the largest score is selected for eviction. In the figure, Mblocks A and B have the highest average score, so they are selected for eviction. Note that overall, MaxAvg is less risky as they evict Mblocks much closer to the LRU than with LRU+. The rationale for MaxAvg is straightforward: it tries to find a group of Mblocks to evict that, on average, is the least recently used group.

In the MaxMin, groups of Mblocks are scored based on their minimum stack distance positions, which is shown in the numbers above the Mblocks. For example, a group of two Mblocks A and B have a minimum score of $\min(4,6)=4$, a group of two Mblocks B and C have a minimum score of $\min(6,1)=1$, etc. After these minimum scores are calculated, the group of Mblocks having the largest score is selected for eviction. In the figure, Mblocks A and B have the highest minimum score, so they are selected for eviction. In this example, MaxMin selects the same Mblocks as MaxAvg for eviction.

We need to have a framework of analysis to understand the replacement policies’ behavior. For this purpose, we look several criteria. The first criteria is whether a replacement policy is self-contained, meaning that it is guaranteed to be able to find a victim. If a replacement policy is self contained, it can be used alone. Otherwise, it has to rely on a different replacement policy as a backup mechanism to break a tie. The second criteria is which Mblocks a replacement policy guarantees not to evict. For example, can a replacement policy evict the MRU Mblock, second MRU Mblock, etc.? This question is important in that it helps us understand the worst case behavior of a replacement policy. An average case behavior is easy to obtain by running programs on a simulation model. However, knowing the replacement policy’s worst case behavior gives us assurance that it will not produce pathological performance cases in some situations that are not covered by existing benchmarks, as well as helps explain poor performance numbers (if any) when running benchmarks on simulation models. Let us discuss these two criteria.

The LRU+ policy is self-contained, because it is always able to find an eviction victim. Starting from the LRU Mblock, it expands to one of two neighboring Mblocks with different stack positions. In contrast, MaxAvg and MaxMin are not self contained as they may produce the same replacement score for different groups of Mblocks, as shown in Figure 2.2: Mblocks C and D have the same MaxAvg score of ‘4’ as Mblocks F and G, Mblocks B and C have the same MaxMin score of ‘1’ as Mblocks E and F, etc. This means that if MaxAvg or MaxMin is deployed, we still need to use a back up mechanism to break the tie. The backup mechanism may be as simple as random, or as sophisticated as using an entirely different replacement policy.

The second criteria of the pathological eviction cases is much harder to evaluate. However, we can prove some properties by constraining our analysis using several simplifying assumptions. Specifically, we are going to assume that there are $N$ Mblocks of the same size residing in one cache line. From these, we will choose a subset of $R$ Mblocks to replace in order to make room for an
incoming Mblock, where \( 1 \leq R < N \). Furthermore, we assume that a cache line wraps around, hence Mblocks on both edges of the cache line can be considered as an eviction group, even though they are not physically contiguous. Under these constrained situations, we will analyze LRU+, MaxAvg, and MaxMin, with regard to what Mblocks are guaranteed not to be selected for eviction.

For LRU+, the following property holds:

**Property 2.3.1** With LRU+, only the MRU Mblock is guaranteed not to be selected for eviction.

The reason why only the MRU Mblock is guaranteed not to be evicted is that when we need to expand the eviction group, we have two Mblocks that can be considered: the left neighbor of the LRU Mblock and the right neighbor of the LRU Mblock. If one of these neighboring Mblocks is the MRU Mblock, then it will not be included into the eviction group, because the other neighboring Mblock will be selected instead. Thus, we can conclude that in the worst case situation, no protection is provided against pathological performance behavior where really useful Mblocks such as the 2nd MRU, 3rd MRU, etc. may be evicted. If we relax the wrap-around cache line assumption, the situation becomes even worse. If the LRU Mblock is at either edge of the cache line, there is only way to expand the eviction group, and if the only neighboring Mblock is the MRU Mblock, it will be selected for eviction.

Let us now consider MaxMin. MaxMin has the following property:

**Property 2.3.2** With MaxMin, the \( \left\lfloor \frac{N-1}{R} \right\rfloor \) most recently used Mblocks will not be selected for eviction.

For example, in a cache line that selects 2-Mblock victim out of 8 Mblocks, \( \left\lfloor \frac{8-1}{2} \right\rfloor = 3 \) of the most recently used Mblocks are guaranteed not to be evicted. For a cache line containing 16 Mblocks \((N = 16)\), 7 of the most recently used Mblocks are guaranteed not to be evicted, if the Mblock group size is 2, or 5 if the group size is 3. To prove it, suppose we start with the MRU Mblock and expand it to contain groups of \( R \) Mblocks. There are \( R \) such possible groups, and each group will have an eviction score of 0, which is the stack position of the MRU Mblock. Next, consider the 2nd MRU block and expand it to contain groups of \( R \) Mblocks. There are \( R \) such possible groups, however some of them may overlap with the ones for the MRU Mblock. The groups that overlap with the MRU Mblock will have a score of 0, but the ones that do not overlap will have a score of 1. Thus, for the 2nd MRU, it adds to anywhere between 1 to \( R \) unique groups. If we repeat the process until the \( k^{th} \) MRU, we have aggregated anywhere between \( R + k - 1 \) groups to \( kR \) groups with the lowest eviction scores. As long as the number of these groups is smaller than \( N \), then there is at least one group that has a higher eviction score. In other words, we need to satisfy the inequality \( kR \leq N - 1 \), equivalent to \( k \leq \frac{N-1}{R} \).

Essentially, MaxMin has a much stronger protection than LRU+. Whereas LRU only guarantees that the MRU Mblock will not be evicted (with wrap-around assumption), MaxMin guarantees
almost half of the most recently used Mblocks from being evicted if the Mblock eviction group size is 2.

Let us now consider MaxAvg. Proving a property with MaxAvg is much more challenging mathematically. However, we can still infer a property of MaxAvg:

**Property 2.3.3** With MaxAvg, only the MRU Mblock is guaranteed not to be evicted when \( R = 2 \), no guarantee otherwise.

To sketch a proof, we start by noticing that the average score for Mblock group itself averages to a constant. Suppose that the LRU stack positions of \( N \) Mblocks are \( a_1, a_2, \ldots, a_N \), where each \( a_i \) may range from 0 (MRU) to \( N-1 \) (LRU) for all \( i \)'s. There are exactly \( N \) potential Mblock group that can be selected for eviction: \( (a_1, a_2, \ldots, a_R), (a_2, a_3, \ldots, a_{R+1}), \ldots, (a_N, a_1, \ldots, a_{R-1}) \). Each of these Mblock group has an average eviction score of \( \frac{1}{R}(a_1 + a_2 + \ldots + a_R), \frac{1}{R}(a_2 + a_3 + \ldots + a_{R+1}), \ldots, \frac{1}{R}(a_N + a_1 + \ldots + a_{R-1}) \).

The average of the eviction scores of Mblock groups turns out to be quite simple: \( \frac{\sum a_i}{R} = \frac{\sum a_i}{N} = \frac{N-1}{2} \).

From here, it follows that any average score that is equal to or larger than \( \frac{N-1}{2} \) may become an eviction victim. Is it possible to arrange \( a_i \)'s such that all Mblock groups have an equal eviction score? It turns out it is not possible, as we are dealing with LRU scores that are whole numbers. The smallest difference in eviction score between two adjacent Mblock group is provably \( \frac{1}{R} \). This means that in order for an Mblock group to be selected as a victim, its eviction score must be at least equal to \( \frac{N-1}{2} + \frac{1}{R} \), the **threshold eviction score**.

In the special case where \( R = 2 \), the threshold eviction score simplifies to \( \frac{N}{2} \). To prove that the MRU Mblock will never be evicted when \( R = 2 \), consider that even when the MRU Mblock (LRU score of 0) neighbors the LRU Mblock (score \( N-1 \)), the eviction score of the MRU and LRU Mblocks is \( \frac{N-1}{2} \), which is less than the threshold eviction score of \( \frac{N}{2} \). Hence, the MRU Mblock is guaranteed to never be evicted. With regard to the 2nd MRU Mblock, its eviction score of 1, when it neighbors the LRU Mblock, the group eviction score is \( \frac{N}{2} \), which meets the threshold eviction score. Thus, the 2nd MRU Mblock may be evicted.

For cases where \( R > 2 \), even the MRU Mblock may get evicted. For example, let us consider \( N = 8, R = 3 \). The MRU Mblock, when it neighbors the LRU (LRU score 7) and 2nd LRU Mblock (LRU score 6), their average eviction score is \( 4 \frac{1}{3} \), which is larger than the threshold eviction score of \( \frac{8-1}{2} + \frac{1}{3} = 3.83 \).

Overall, in the worst case, the MaxAvg and LRU+ provide weak protection from evicting the recently used Mblocks. In contrast, MaxMin provides the strongest protection, since several MRU Mblocks will not be evicted, depending on \( N \) and \( R \).

### 2.3.2 Placement Fit Policy

Related to the replacement policy is a question of whether we should take into account the fit when considering Mblocks to evict. Since the Mblock placement is similar to the memory allocation, we
can use traditional memory allocation policies such as best fit and worst fit. However, unlike the memory allocation problem, the size of free space depends on the size of victim Mblocks chosen by replacement policy. Thus, in many cases the placement fit policy is only usable for breaking the tie between several victim Mblock groups with the same replacement scores.

2.3.3 Dealing with Block Miss

Besides of the placement/replacement policy required when we have an Mblock miss, there is another type of DFP cache miss: a block miss. This is a situation where the Mblock being accessed is found in the DFP cache, but the block being accessed is not present in the Mblock. This situation occurs when an Mblock access pattern changes that the block is not anticipated by the footprint predictor. The footprint predictor predicts blocks that are useful based on the Mblock's access pattern in the past. However, when the access pattern of the Mblock changes, the processor may access a block that was not accessed in the past and a block miss occurs.

When there is a block miss, the block must be inserted into the Mblock. How this insertion is handled depends on whether there is a hole to accommodate the new block or not. Depending on the location of the block insertion, there are several cases to consider. A hole may be there from fragmentation created by a previous Mblock allocation. For example, if a three-block Mblock was evicted to make room for a two-block Mblock, then a hole is created in its place. Figure 2.3 (top diagram) illustrates the case where there is a hole to accommodate the block. The figure shows two Mblocks (E and F), each having three blocks, co-located on a single cache line. There are two holes in the cache line, one between the two Mblocks. Suppose that the last block in Mblock E is accessed and is missed. In this case, the Mblock E reclaims the hole, and expands to accommodate the new block. The valid bits are updated accordingly.

Figure 2.3 (middle diagram) depicts a different case involving Mblocks G and H. In this case, there is no hole for the Mblock G to expand, hence, Mblock H needs to be removed and re-inserted at a different location to make room for the new block. The valid bits of Mblock G and the starting position of Mblock H are updated accordingly. This case is more expensive than the first one, since re-insertion requires reading a large number of bytes from a row for one block miss and writing them back to the row when the row is closed. However, we found that it does not happen often due to high footprint predictor's accuracy. Furthermore, the overhead itself is very minor because it is overlapped with the miss latency; hence there is no critical-path delay. Likewise, if the block being missed is in the middle of the Mblock, the entire Mblock is removed and then re-inserted.

Figure 2.3 (bottom diagram) depicts another case involving Mblocks I and J. In this case, there is no hole anywhere in the cache line for the Mblock I to expand, hence, an Mblock needs to be evicted to make room. In the figure, Mblock J is evicted, and this creates space for the new block along with introducing new holes. Note that in this case, the replacement policy can be invoked.
Figure 2.3 Block miss handling in DFC.

and this situation can be handled as if there is an Mblock miss. This case is potentially the most costly because it results in an Mblock being evicted. However, to avoid the complexity and latency for handling a full Mblock miss, and counting on the low number of occurrences of this case, we simply choose to evict the Mblock that gets in the way for the Mblock that needs to expand.

2.3.4 Early Footprint Update

So far we have discussed how an Mblock and block miss is handled. We will now discuss another aspect of the DFC, which is when the footprint predictor is updated. Our DFP predictor shares a similarity with the original footprint predictor [Jev13], but differs in a key aspect. The DFP predictor is similar in the way the footprint predictor is indexed, using a combination of a part of the memory
address (spatial offset) and program counter. Using the PC and spatial offset helps in getting a footprint prediction for highly reused code that works on different data items yet exhibiting the same access pattern. Some examples of such a scenario includes database scan and join operations [Jev13].

In the Footprint cache and our basic DFC design, the footprint of an Mblock is updated when the Mblock is evicted from the cache. Blocks that were accessed during the Mblock’s residency in the DFC became the footprint for the Mblock, causing them to be fetched the next time the cache misses on the Mblock. However, if the cache size grows, Mblocks stay longer in the cache and are evicted less frequently, which delays or even avoids footprint updates. This slows down learning for the footprint predictor, and reduces its prediction coverage. Especially, if we consider a scenario where data access pattern due to code reuse is applied over a large data set, this slow footprint learning reduces the opportunities for making footprint prediction. To overcome this slow footprint learning, we propose *early footprint update* (EFU), which allows footprints to be updated even when an Mblock is not evicted from the cache yet. One naive way to update the footprint prediction is whenever there is a new block that is accessed in the Mblock. However, this will result in too frequent footprint table updates, which increases occupancy and power consumption of the FHT. Furthermore, updating the FHT too soon is risky because incomplete footprints reduces footprint prediction accuracy. Thus, we should design footprint update criteria that is not too aggressive and yet still allow early updates.

To achieve that goal, the criteria for an early update that we use is the number of accessed blocks in an Mblock during its residency in the DFC. Over time, during an Mblock’s residency in the cache, more and more of its blocks are accessed. Once the number of blocks accessed reaches a threshold, the footprint is updated. We refer to this as Early Footprint Update (EFU). This allows the footprint predictor to learn footprints early, increasing the footprint prediction coverage. However, the footprint predictor is possibly updated using incomplete access pattern information, hence EFU trades off accuracy for coverage. As long as the benefit from increased coverage outweighs the drawback from reduced accuracy, EFU improves performance. Through evaluation, we found that using multiple threshold values works well. For example, we may have threshold values of 16, 24, and 32. When the number of accessed blocks reaches 16 for an Mblock, we update the footprint predictor. Later, when the number of accessed blocks reaches 24, we re-update the footprint predictor, etc.

2.3.5 Overhead analysis

Cache designs normally have a straightforward way to locate a cache block from its physical address. However, in order to access a specific block in DFC, it requires to calculate the exact bit location and counting the number of preceding ones in a bit vector in tag array. This additional computation requires 4 levels of parallel adders along with a masking logic to mask succeeding bits after the bit location [MF02]. Considering the inverter delay as 13.9ps by 2007 paper [Arn07], 4 adders with one
masking logic is roughly less than 0.6 ns. We believe that such logic is probably going to cost way less than half nanosecond if we consider the technology scaling. Thus, a cache block look up overhead is negligible.

While the replacement policies discussed here are more complex than a regular LRU, fortunately it is only used for the LLC which is much less frequently accessed than other cache levels. Moreover, the victim selection itself will be very fast, because all operations can be performed in parallel. The victim selection starts with making victim Mblock groups, where all victim Mblock groups can be made in one time SRAM access in parallel with a few additions and comparisons in case of MaxMin and LRU+. Later on, looking up the victim among groups can be executed in parallel as well, then the smallest group satisfying the incoming macro block size is selected. Thus, only one time SRAM access to every macro block tag in the set will be dominant time and all other operations are processed in parallel. Furthermore, the replacement policy computation can be overlapped with the LLC miss latency, so that it does not generate any critical-path delay.

2.4 Methodology

For evaluating DFC, we primarily target scale-out memory intensive server applications/benchmarks from CloudSuite 2.0 [Fer12], including Data Analytics (DA), Data Caching (DC), Data Serving (DS), Software Testing (ST), Web Search (WS), and Graph Analysis (GA). All these benchmarks are run on a Simics 3.0.31 virtual machine with AMD64 CPU and 16GB main memory, running Linux OS Ubuntu 10. We run each server benchmark for about a few hours until each server benchmark enters the phase of processing client's requests. Then, we use 3 billion memory instructions to produce a trace. In the simulation, 2 billion memory instructions are used to warm up the cache structures and 1 billion memory instructions are used for detailed simulation. The trace is designed to be sufficiently comprehensive to enable timing and power simulation. It contains basic information such as the memory instruction type, program counter, and address. The trace also contains dependence information between memory instructions, the number of non-memory instructions separating two memory instructions broken into ones (directly or indirectly) dependent on the preceding memory instruction and ones that are independent.

The traces are then used for simulation of a baseline machine model shown in Table 2.1. We model a detailed memory hierarchy model, and off-chip and stacked DRAM timing model using DRAMSim2 [Ros11]. To estimate execution time, we implement an abstract Tomasulo pipeline. The pipeline has a 128-entry instruction window. An instruction is entered into the window as it is issued, and is executed once the instruction it depends on has completed execution. All instructions are assumed to take one clock cycle to execute, except for memory instructions of which their execution times depend on the hit/miss outcome at various caches and DRAMs. Therefore, while simple, our pipeline model takes into account both memory data dependence stalls and structural hazard stalls.
We envision a 16-core system with SRAM L1 and L2 caches private to each core, and a stacked DRAM L3 cache shared by all cores. Since our trace was collected from a single core, we assume the L3 cache to be partitioned among all cores equally, and simulate just one core and one partition of the L3 cache. The main memory is assumed to be off-chip DRAM. DRAM parameters for both the L3 cache and the off-chip DRAMs are shown in the table.

Table 2.1 The baseline system configuration.

<table>
<thead>
<tr>
<th>Processor</th>
<th>16 cores 3.2GHz, 1 IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 I$ and D$</td>
<td>32KB, 4-ways, 64B Block</td>
</tr>
<tr>
<td></td>
<td>2 cycles each</td>
</tr>
<tr>
<td>L2 cache</td>
<td>Variable size and latency (Table ??)</td>
</tr>
<tr>
<td>Off-chip DRAM</td>
<td>16GB, DDR3-1600 (800MHz), 1 channel</td>
</tr>
<tr>
<td></td>
<td>8 Banks per rank, 16KB row-buffer</td>
</tr>
<tr>
<td>Stacked DRAM</td>
<td>DDR-like interface (1600MHz), 4 channels</td>
</tr>
<tr>
<td></td>
<td>16 Banks per rank, 8KB row-buffer</td>
</tr>
<tr>
<td></td>
<td>128 bits per channel</td>
</tr>
<tr>
<td>$t_{CAS}$-$t_{RCD}$-$t_{RP}$-$t_{RAS}$-$t_{RC}$</td>
<td>11-11-28-39</td>
</tr>
<tr>
<td>$t_{WR}$-$t_{WTR}$-$t_{RP}$-$t_{RD}$-$t_{FAW}$</td>
<td>12-6-5-24</td>
</tr>
</tbody>
</table>

For the DRAM L3 (LLC) cache, we implement six different schemes: Footprint cache [Jev13], Unison cache [Jev14], block-based DRAM cache (i.e. DRAM cache with a traditional 64-byte block size), an oracle DRAM cache that always hits and has no tag space overheads, our scheme DFC, and for comparison purpose, no L3 cache. For Footprint cache and Unison cache, we follow the optimized parameters from [Jev14], where the cache is 4-way set associative, each DRAM row is 8KB, each Mblock is 2KB for Footprint cache, and each Mblock is 960B for Unison cache. For DFC, we implement a design with 8KB DRAM row which can hold anywhere from four Mblocks to eight Mblocks.

To have a fair comparison, we use a fixed SRAM budget of 16MB (1MB per core), which can be allocated in different proportions between the L2 cache and the tag array of the L3 stacked DRAM cache. As a result of the constant SRAM budget, the L2 cache capacity and latency and the L3 tag capacity and latency vary for different techniques, as shown in Table ??'. As shown in the table, we tested all cache capacities from 32MB to 512MB (2MB to 32MB per core). For the block-based stacked DRAM L3 cache, its high tag space requirement caps the L2 cache capacity to 16MB per core.
Table 2.2 L2 cache size configurations as affected by a limited SRAM budget.

<table>
<thead>
<tr>
<th>LLC size (MB)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unison</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Size (MB)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Footprint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Size (MB)</td>
<td>15.80</td>
<td>15.59</td>
<td>15.22</td>
<td>14.45</td>
<td>12.94</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>DFC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Size (MB)</td>
<td>15.58</td>
<td>15.16</td>
<td>14.34</td>
<td>12.73</td>
<td>9.55</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>64B Based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Size (MB)</td>
<td>13.94</td>
<td>12</td>
<td>8.25</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>Latency (cycles)</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>None</td>
</tr>
</tbody>
</table>

2.5 Evaluation

2.5.1 Overall Performance

Figure 2.4 shows the speedup ratio of Footprint cache (Footprint), Unison cache (Unison), our design (DFC with MaxMin), and 64-byte block size cache (64B block), when the L3 cache capacity per core is 64B (part (a)) or 256MB (part (b)). For reference, an oracle L3 cache with zero miss rate and zero tag overheads (Full Stacked DRAM) is also added to both figures. The speedups are relative to a baseline of a system with only off-chip DRAM.

Let us examine the 64MB L3 cache first. Here we can see that the oracle L3 cache can achieve a geometric mean (gmean) of speedups of 19%, but the speedups decrease to 8.8% when the L3 cache size is limited at 64MB with the 64-byte block size. Note that Footprint cache performs better than Unison cache, consistent with results reported in the Unison cache paper [Jev14]. Although the way predictor reduces its hit latency, it shows that it still suffers from its high miss latency. According to Unison cache paper, it starts to perform better than Footprint cache in cache size larger than 1GB. However, considering the diminishing returns of larger caches, we believe that such large cache is not practical and leaving the reminder of the stacked DRAM for part of main memory is a more viable choice. With a small block size, at 64MB, the SRAM budget needed for the L3 tag array only reduces the L2 cache size slightly, from 16MB to 12MB, so it produces an acceptable performance. The Footprint cache produces speedups in some applications but slowdowns in others, leading to a gmean speedup of 3.4%. Footprint cache produces a significant slowdown in graph analysis (GA) because only a few blocks in an MBlock are useful; in Footprint cache, such an Mblock is mostly empty but still occupies a 2KB space in the cache. Such internal fragmentation of Mblocks is eliminated in DFC, hence DFC performs much better than Footprint cache for this application.
In contrast to the Footprint cache, DFC has no internal fragmentation. However, DFC may incur some external fragmentation when there are holes in a cache line that are not sufficiently large to an additional Mblock. 64-byte block cache has no fragmentation whatsoever, so it slightly outperforms DFC in few benchmarks (ST, DS, and WS). However, DFC outperforms 64-byte block cache overall (9.4% vs. 8.8% gmean). Both significantly outperform the Footprint cache because of their efficient use of L3 cache capacity.

Moving on to the 256MB L3 cache, the situation changes drastically. The 64-byte block L3 cache incurs much larger SRAM budget for the tag array (leaving the L2 cache with 1MB), resulting in
slowdowns for half of the applications (DC, DS, and WS) and it performs worse than baseline overall. In contrast, DFC achieves 13.2% gmean speedup, and even reaching 20% for DA and DS. With 256MB, the L3 cache capacity is large enough to hold much of the working sets of the applications, such that even inefficient use of cache capacity in the Footprint cache allows it to perform quite well (11.1% gmean speedup).

In what seems like an anomaly in DA, DFC performs very slightly better than oracle. First, the L3 cache hit rates are already very high in DFC, hence there are not many L3 cache misses. Second, because the DFC is much smaller and has much fewer DRAM rows, there are cases where two Mblocks mapping to a single DRAM row, leading to more row buffer hits, and better performance in DFC.

Table 2.3 L3 miss ratios of Footprint cache (FC) vs. our DFC. All numbers are percentages.

<table>
<thead>
<tr>
<th>App</th>
<th>64MB</th>
<th>256MB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FC</td>
<td>DFC</td>
</tr>
<tr>
<td>DA</td>
<td>41.7</td>
<td>29.8</td>
</tr>
<tr>
<td>DC</td>
<td>34.9</td>
<td>19.0</td>
</tr>
<tr>
<td>GA</td>
<td>85.7</td>
<td>83.6</td>
</tr>
<tr>
<td>ST</td>
<td>48.2</td>
<td>41.5</td>
</tr>
<tr>
<td>DS</td>
<td>58.9</td>
<td>37.8</td>
</tr>
<tr>
<td>WS</td>
<td>53.1</td>
<td>45.5</td>
</tr>
<tr>
<td>Avg</td>
<td>10.9</td>
<td></td>
</tr>
</tbody>
</table>

Let us compare DFC vs. Footprint cache to understand better in what ways they differ. We will look at the cache miss ratio, footprint prediction accuracy and coverage. Table 2.3 shows the L3 cache miss ratios of Footprint cache vs. DFC. The data shows that DFC always achieve lower miss ratios vs. Footprint cache, achieving an average 10.9% better for 64MB L3 cache. The spread decreases to only 6.5% when the L3 cache size is 256MB, because more of the working sets of the applications fit in the L3 cache, reducing the capacity pressure that can be relieved by DFC.

The accuracy of Footprint cache and DFC prediction is shown in Table ???. The table shows accuracy (number of blocks of an Mblock fetched due to footprint prediction that are subsequently used by the processor divided by total blocks fetched), and under-prediction rate (number of blocks that are not fetched that are subsequently missed by the processor divided by total number of blocks fetched). Ideally, we want high accuracy and low under-prediction. Over-prediction (fetching too many blocks) is costly for DFC because unused blocks consume bandwidth and L3 cache space, but is only slightly costly for Footprint cache as they do not change the L3 cache space usage. However, under-prediction is very costly for both schemes because if the processor accesses a block that has not been predicted useful, even when the Mblock is present, the block has to be fetched from the
Table 2.4  Predictor accuracy comparison of Footprint cache vs DFC. All numbers are percentages.

<table>
<thead>
<tr>
<th></th>
<th>Footprint cache</th>
<th>DFC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accur</td>
<td>Under</td>
</tr>
<tr>
<td>DA</td>
<td>92.7</td>
<td>14.4</td>
</tr>
<tr>
<td>DC</td>
<td>70.1</td>
<td>38.2</td>
</tr>
<tr>
<td>GA</td>
<td>25.3</td>
<td>78.9</td>
</tr>
<tr>
<td>ST</td>
<td>69.4</td>
<td>37.5</td>
</tr>
<tr>
<td>DS</td>
<td>71.5</td>
<td>51.8</td>
</tr>
<tr>
<td>WS</td>
<td>69.1</td>
<td>47.3</td>
</tr>
<tr>
<td></td>
<td>66.4</td>
<td>44.7</td>
</tr>
</tbody>
</table>

off-chip DRAM with great latencies. Thus, under-prediction is critical to performance. The table shows that DFC achieves about half the under-prediction rate as Footprint cache (44.7% vs. 24.7%). One reason for these excellent numbers for DFC is that Mblocks stay longer in the L3 cache due to more efficient usage of cache capacity, and thus have a much higher chance of capturing the entire footprint behavior of an Mblock. To illustrate an example, suppose the first half of an Mblock is accessed first, and the second half of an Mblock is accessed last. If during this time the Mblock resides in the cache, when the Mblock is evicted, its footprint is accurately captured and reflects all the blocks that were accessed. However, if the Mblock only resides for half as long, when it is evicted, its footprint only reflects one half of the access, and the next time it is brought in and evicted, the footprint reflects the other half. Thus, footprint learning and prediction is more stable and more accurate in DFC.

**Energy consumption.** For energy simulation, we measured the energy consumption from L2 cache, off-chip DRAM, and stacked DRAM and sum them. Compared to Footprint cache, DFC reduces the energy consumption from 0.8% to -22.5% at 64MB depending on benchmarks and average -4.4%. Most of energy saving originates from reducing off-chip memory traffic since DFC keeps Mblocks in the cache longer and hence incur fewer cache misses. A secondary effect comes from DFC’s higher footprint prediction accuracy which increases row buffer hit rate and reduces the number of DRAM row activations, which are the highest energy required operation [Jev13; Jev14; Vol14]. However, the energy gains are decreased to -0.1% at 256MB dram cache, which is due to increased SRAM cache misses in DFC and narrowed DRAM cache miss difference between Footprint cache and DFC.

### 2.5.2 Replacement Policy Comparison

Figure 2.5 compares the LRU+, MaxAvg, and MaxMin replacement policies for a 64MB DFC in terms of miss ratios (a) and speedups over Footprint cache (b). The figure shows that replacement policies show comparable performance for DC, GA, and ST; but produce different performance for DA, DS, and WS. When the performance of different policies differ, MaxMin achieves the best performance.
Figure 2.5 Comparing LRU+, MaxAvg, and MaxMin replacement policies for a 64MB DFC in terms of miss ratios (a) and speedups over Footprint cache (b).

(lowest miss ratios and highest speedups), followed by MaxAvg and finally by LRU+. As we discussed in Section 2.3.1, MaxMin exhibits the strongest guarantee of not evicting nearly half of the most recently used Mblocks, whereas LRU+ and MaxAvg do not even guarantee that the MRU Mblock will not be evicted. To investigate why MaxAvg slightly outperforms LRU+, we collect statistics on the number of times the MRU Mblock gets evicted. The statistics shows that MaxAvg evicts the MRU Mblock only 12.3–19.1% of the number of MRU Mblock evictions performed by LRU+. It also confirms that MaxMin never evicted MRU Mblock in any of these benchmarks.
2.5.3 Placement Policy

When an Mblock needs to be allocated in the cache, and several candidate groups of Mblocks have equal replacement scores, we can break the tie by using the best fit or worst fit policy. In worst fit, the largest Mblock group is evicted so that even when a part of the freed capacity is used for the incoming Mblock, the leftover portion may be large enough for allocation of another Mblock. In storage allocation literature, best fit is cited as leaving a lot of small segments that are not usable, such that worst fit fares better. In DFC, however, blocks evicted by worst fit may have to be refetched, incurring significant memory bandwidth and latency to refetch them. We test these policies and found that worst fit performs very slightly worse: it shows up to 4% higher miss ratio compared to best fit.

2.5.4 Early Footprint Update

![Miss-ratio Comparison for Early Footprint Update](image)

Figure 2.6 Miss-ratio Comparison for Early Footprint Update

For the evaluation, we set up three early update thresholds, 16 blocks, 24 blocks, and 32 blocks. At every time the number of blocks in Mblock reaches one of thresholds, the Mblock’s footprint is updated in the footprint history table (FHT). Figure 2.6 shows the miss ratio improvement of DFC with EFU compared to DFC without EFU in 64MB stacked DRAM. Because the trend is almost similar in all three replacement policies, we simply select MaxMin to draw the graph. EFU helps to expedite footprint updates which can be delayed by the large DRAM size. Therefore, if the L3 size is large enough to slow down footprint updates, EFU can increase prediction coverage, which leads to lower miss ratio. From the figure, we can see that 0–2.2% reduction in miss ratio.
To investigate further, we measure footprint coverage (number of Mblock misses that can be predicted by the footprint predictor). With the EFU, we expect increased coverage. We found that coverage without EFU is already nearly 100% for DC, GA, and ST; hence EFU does not make much differences. However, for DA, DS, and especially WS, the footprint coverage increases measurably (2.1%, 2.6%, and 8.4%, respectively).

2.6 Sensitivity Study

![Figure 2.7 Performance Comparison over number of tags difference per set](image)

**Sensitivity to Additional Tags.** In DFC, the number of tags should be carefully decided based on the SRAM budget because its performance is highly related to it. If DFC has too many tags, it can keep more Mblocks but also requires larger SRAM space for the additional tags that might not be used. In contrast, if the number of tags is too small, the L2 cache can be made larger, but the DFC can keep fewer Mblocks. Thus, we evaluated the impact of tag counts per set. Figure 2.7 depicts the impact of tag count per set. Footprint cache has a fixed 4 tags per set, while for DFC we can vary the number of tags per set. The figure shows the speedups for different L3 cache size and number of tags, averaged over all benchmarks. We can make several conclusions. First, 8 tags performs the best overall across all benchmarks, especially for larger DFC. For smaller DFC, the impact from tag array overhead is small, so it favors 10 tags per set. Note that this means that overall, DFC can keep between up to $2 - 2.5\times$ more Mblocks than Footprint cache.

**Sensitivity to DRAM Speed.** DRAM can process only one request in one channel. In our previous experiments, we expected that there is no contention in DRAM and only divide DRAM capacities by
cores. However, if the number of cores grows, the request contention happens and a request should wait until its previous request is processed. Under heavily contended circumstances, we expect that DFC performs better because stacked DRAM handles the larger fraction of traffic compared to off-chip DRAM. We simulate the impact of DRAM contention by increasing tRC, the time interval between accesses to different rows in a bank. Footprint predictor issues burst of memory operations, but these are not affected by this limitation, because these requests affect the same DRAM row.

Figure 2.8 shows the speedups of DFC normalized to Footprint cache. The figure shows that as the DRAM’s tRC increases, DFC’s speedups relative to Footprint Cache increase. This is expected, because DFC reduces the number of off-chip memory accesses.
In Chapter 3 we propose the speculative execution on NVMM, calling speculative persistence. The remainder of the paper is organized as follows. Section 3.1 provides background on memory persistency, and describes the new PMEM instructions. Section 3.2 describes the new workloads we created using write-ahead logging, Section 3.3 describes our new architecture and its implementation, Section 3.4 describes the evaluation methodology, and Section 3.5 evaluates our design and presents our key findings. Finally, Section 3.6 discusses related work to speculation.

3.1 Background

3.1.1 Memory Persistency Models

A modern processor design relies on multiple levels of caches and out-of-order instruction execution. In such a system, the order in which writes are made durable to the NVMM is unpredictable and does not follow program order, as it depends on the order of write backs from the last level cache (LLC) to the NVMM. Upon failure, values in the cache (which may correspond to older stores) are lost,
while values in the NVMM (which may correspond to younger stores) are not. To achieve failure safety, programmers need precise specification of when stores will be durable in the NVMM, referred to as the persistency model, in order to reason about how they need to rewrite their software to achieve failure safety of their code.

Previous researches proposed several memory persistency models, strict persistency, epoch persistency, buffered epoch persistency, strand persistency, and transactional persistency. Strict persistency [Pel14] piggybacks the sequential consistency model, by specifying that a store that is globally visible must also have persisted in NVMM. Due to this constraint, before each store persists to NVMM, all previous stores must have persisted to NVMM. The easiest way to implement strict persistency is using a write through cache hierarchy for stores. Since any visible stores are persisted, failure safety reasoning is easier for programmers. However, it comes with significant performance costs of not allowing write reordering and write coalescing that naturally occur in a write back caches.

Epoch persistency relaxes the ordering constraint in strict persistency [Con09; Pel14]. Epoch persistency allows programmer to put persist barrier which defines an epoch in the program. Stores from an epoch (i.e. between two persist barriers) can persist in any order, but they must all persist at the persist barrier. Write coalescing can occur for stores from the same epoch. At the persist barrier, the processor may stall waiting for all stores from the epoch to persist. In a related buffered epoch persistency model [Con09; Jos15], a persist barrier does not force prior stores to persist right away (hence the processor may not stall), as long as stores from one epoch persisted prior to any stores from the next epoch.

Epoch persistency only relaxes the ordering of an epoch of stores from a thread. Strand persistency [Pel14] expands this to groups of stores from multiple threads, defined by strand barriers.

Transactional memory persistency relies on transactions as a unit of persistency [Kol16b], where all stores in a transaction either persist together or not at all. The study also proposed using an extra buffer and cache bypassing for persist in every core as performance optimizations. Copy-on-write can be another way to implement a transaction with checkpoints and Rei et al. adopt it in their proposed systems using NV memory for checkpoint [Ren15].

### 3.1.2 Intel PMEM Persistency Model

Intel recently announced a new PMEM persistency model in pmem.io [Int16d], based on several new instructions, such as `clwb`, `clflushopt`, and `pcommit`, used in conjunction with existing x86 instructions such as `clflush` and `sfence`.

Compared to other persistency models, PMEM instructions provide a more flexible approach towards memory persistency. It allows programmers to construct other persistency models such as

---

1We are aware that Intel has deprecated `pcommit` [Int16a]. In our work, we do not assume that the memory controller is part of the persistency domain, so `pcommit` is still required.
strict and epoch persistency (if desired), but it also allows programmers to select which stores need to persist and in which order they persist.

Figure 3.1 illustrates the PMEM instruction operations. The \texttt{clflushopt} and \texttt{clwb} instructions clean dirty cache blocks from the cache hierarchy by writing them back, and in the case of \texttt{clflushopt}, the blocks are also evicted. After dirty data is evicted from the cache, it may be waiting in a buffer in a memory controller. The dirty data is not durable yet until it is removed from the buffer and written to the NVMM. The \texttt{pcommit} instruction is used to force durability of such pending writebacks, by forcing the memory controllers to flush all pending writes. When all buffers have been flushed, each memory controller sends an acknowledgement back to the core that executed the \texttt{pcommit}, indicating the completion of \texttt{pcommit}.

By appending \texttt{clflushopt} or \texttt{clwb}, and \texttt{pcommit} after stores that need to persist, programmers can selectively choose which stores should persist, and when they persist. This flexibility allows programmers to persist only stores to recovery-critical data, instead of stores to all data. It also permits write coalescing. These features are critical to achieving low overhead failure safety.

However, using PMEM instructions (\texttt{clflushopt}, \texttt{clwb}, and \texttt{pcommit}) by themselves is not sufficient because there is no ordering guarantee with respect to other stores or other persistence instructions, except for some implicit dependences on the same address. \texttt{Clflushopt} is ordered

\footnote{\texttt{clflush} has a similar functionality but much worse performance, so we do not use it in our study.}
with respect to other stores on the same cache block, however clwb is only ordered with respect to stores to the same address. Otherwise, to force an ordering among two of these instructions, a store fence instruction (sfence) is required. Sfence, which was originally an instruction used for memory consistency, takes up an additional role in that when placed after PMEM instructions, it will wait for all pending PMEM instructions to complete before retiring, and prevent following stores and PMEM instructions from executing until it completes.

To illustrate an example of the usage of the PMEM instructions, suppose that we wish to persist a store to address X before modifying address Y. Then, we would write:

\[
i1: \text{st X, 1};
\]
\[
i2: \text{clwb X};
\]
\[
i3: \text{sfence};
\]
\[
i4: \text{pcommit};
\]
\[
i5: \text{sfence};
\]
\[
i6: \text{st Y, 1};
\]

Instruction i2 (clwb) forces the store X to be written back from the cache, instruction i4 (pcommit) persists the store to the NVMM. The first sfence (i3) stalls the pcommit until the write back is complete. Next, the second sfence (i5) makes the update to X durable, and forces store Y (i6) to wait until the data block containing X is truly durable. pcommit's completion is detected when the write buffers in the memory controller are flushed and the processor has received acknowledgement from all memory controllers. Thus, pcommit may take a long time to complete, and becomes the major performance bottleneck, which we strive to address in this chapter.

In our speculative persistence, the processor takes a checkpoint and enters speculative region, and then speculatively retires the sfence instruction i5, and proceeds executing instruction i6 and beyond. The processor exits speculative region when the pcommit completes successfully.

### 3.2 Workloads

In the previous section, we have discussed PMEM instructions that make up a flexible building block of memory persistency that programmers can control. In this section, we discuss how we utilized the instructions to develop the workloads that we use in our evaluation. We focus on single-threaded data structures and algorithms commonly used in databases and file systems. To achieve failure safety, we use a write-ahead logging transaction approach. We wish to investigate failure safety performance issues for single-threaded programs, and leave multi-threaded programs for future work.
3.2.1 Failure Safety through Transactions Using Write-Ahead Logging

Failure safe updates to non-volatile storage have long been achieved through transactions [Moh92]. We believe that future software must update NVMM using transactions as well. It may be implemented directly in software by programmers or provided through a compiler or library [Vol11]. We implement transactions directly in software as part of the normal operation of data structures using a form of write-ahead logging [Moh92].

The general strategy of write-ahead logging is to make an undo log of all desired changes to memory before making any modifications. That way if a failure occurs in the middle of the update, then the undo log can be used for recovery. The general sequence of steps to complete the transaction on NVM are as follows:

1. Perform undo-logging. Make the undo-log updates durable.
2. Logging_bit is set and made durable, indicating a transaction has begun.
3. Commit updates to the memory and make them durable.
4. Logging_bit is unset and made durable, indicating the transaction is complete.

To understand how this works, we must consider the state of memory after a failure. The logging_bit indicates whether a transaction is in progress or not. If the logging_bit is 0, no transaction was in progress when the failure occurred and the current data structure is reliable. Otherwise, the logging_bit is 1, and a transaction was in the middle of processing. In this case, the undo log will be used to recover the state of the data structure. Because we do not know at what step the failure occurred, we must pessimistically recover using the undo log regardless.

To ensure correctness of the above, the 4 steps must be strictly ordered. If Step 2 begins before Step 1 completed, the premature undo log may be erroneously applied to the data structure. Likewise, if commits began before the logging_bit were set, then an incomplete set of updates may go undetected. We can enforce the necessary ordering between these steps using persist barriers, the sfence-pcommit-sfence discussed in the previous section. Given that each step needs a persist barrier, this implies that at least 4 pcommits and 8 sfence operations are needed per transactional update to NVM. Also, compared to a volatile data structure, the added cost of undo-logging will also be a significant overhead.

3.2.1.1 Detailed Example: A Non-Volatile Linked List

Figure 3.2 shows an example of our transactional code for a linked list. We choose a linked list code for illustration due to its simplicity (e.g., compared to balanced trees). Before making any updates in the linked list, the modified nodes need to be logged. In the example, a new node ‘temp’ needs to be inserted after node ‘nn’. Hence, we log data of node ‘nn’ and the address of ‘nn’. After the logging
is completed, a logging_bit is set. If the system crashes during the transaction, if the logging_bit is set, the transaction is undone by overwriting the original location using the logged data. If the bit is unset, the data structure is consistent as modifications to the linked list have not occurred. The example in the figure shows that one transaction requires four pcommits with several sfences and clwb. Since the data structure is stored in the NVMM, a system crash may result in the inconsistent data structure. In Figure 3.2, if the system crash happens after line 28 but before line 29, the data structure becomes inconsistent, resulting in the linked list shown in Figure 3.3. This illustrates
the importance of the use of a transactional approach, which in our case is supported through write-ahead logging.

### 3.2.2 Workload Construction

Using the write-ahead logging approach discussed above, implemented using PMEM persistency instructions, we constructed a workload consisting of benchmarks with data structures listed in the Table ??.

#### Table 3.1

Benchmarks constructed for our study. For all benchmarks, we size each node to be 64 bytes and align them to cache blocks. Thus, to persist one node update, one clwb will be required.

<table>
<thead>
<tr>
<th>Benchmark (Abbrev.)</th>
<th>Description</th>
<th>#InitOps</th>
<th>#SimOps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph (GH)</td>
<td>Insert or delete edges in a graph</td>
<td>2600000</td>
<td>100000</td>
</tr>
<tr>
<td>HashMap (HM)</td>
<td>Insert or delete entries in a hash map</td>
<td>1500000</td>
<td>100000</td>
</tr>
<tr>
<td>Linked List (LL)</td>
<td>Insert or delete nodes in a linked list (Max:1024)</td>
<td>500</td>
<td>50000</td>
</tr>
<tr>
<td>String Swap (SS)</td>
<td>Swap strings in a string array</td>
<td>1200000</td>
<td>500000</td>
</tr>
<tr>
<td>AVL tree (AT)</td>
<td>Insert or delete nodes in an AVL tree</td>
<td>1000000</td>
<td>50000</td>
</tr>
<tr>
<td>B tree (BT)</td>
<td>Insert or delete nodes in a B tree</td>
<td>1000000</td>
<td>50000</td>
</tr>
<tr>
<td>RB tree (RT)</td>
<td>Insert or delete nodes in an RB tree</td>
<td>1500000</td>
<td>50000</td>
</tr>
</tbody>
</table>
We found that there are two types of data structures depending on the number of nodes involved in update operations. Hashmap, linked list, and graph belong to a first type of benchmarks with few nodes involved in an update operation. They have very small overheads from the undo logging code. In Hashmap, an operation uses a hash function to map a key to an index to a hash table entry. We use chained collision policy, so that if the entry is already populated, the next consecutive entry is checked, and so on. Once a free entry is found, the operation logs the location and the size of the hash table, before inserting the new record. If the key is found, the entry is logged before deleting the record from the entry. As discussed earlier, after logging, logging_bit is set to 1 and once the insertion or deletion is complete, the bit is reset. In Hashmap, if no free entry is found for insertion, the table is resized. In this case, we create a new table twice the size of the original table, and move all records from the original table to the new one. During record copying, each insertion is followed by clwb. pcommit persists the completion of the resizing. Other data structures, LinkedList and Graph are implemented in a similar manner, but without data structure resizing and copying. In the case of String Swap, we get two random indexes to swap in a string array. The length of each string in the entry is 256. Before swapping, an operation undo logs two strings in an indexed entry. After the logging, eight clwbs are issued for logging entries and one clwb is for indexes. After the swap is completed, another eight clwbs are issued along with pcommit.

Self-balancing trees such as AVL tree, B tree, and RB tree, belong to a second type of benchmarks with a variable numbers of nodes involved in each update operation. Due to tree rebalancing, logging becomes more complex and has high overheads. Sometimes, all nodes in a root-to-leaf path are involved in an operation.

In a self-balancing tree, after an insertion or deletion of a node, the tree balance property is checked. If the tree is no longer balanced, rebalancing is triggered. Here, we face a design choice of how to transactionalize the tree balancing to achieve failure safety: incremental logging or full logging.

Incremental logging breaks rebalancing into multiple steps, where in each step we log as few nodes as needed to perform balancing for a particular affected node. In many cases, each rebalancing only involves one or two levels. In some cases, the rebalancing is escalated to higher levels of the trees. Hence, the nodes involved in each rebalancing step can be made a small logging unit. Figure 3.4 illustrates the incremental logging approach on a 2-3 Btree. A 2-3 Btree is a sorted balanced tree where each non-leaf node can have anywhere between two and three children nodes. Data is stored in the leaf nodes, while non-leaf nodes store keys to accelerate searching. In order to insert a new node, the tree is traversed recursively from the root to the appropriate leaf based on the new node's key.

The top diagram of the figure shows a leaf node 4 is about to be inserted into the tree, which will result in the node (1,3) with too many children, which will require node (1,3) to be split to keep the tree balanced. In the incremental logging approach, node (1,3) is logged prior to insertion. After leaf
node 4 is inserted, rebalancing is triggered. Next, node (5,9) is logged and node (1,3) is split. This rebalancing is escalated until the tree is balanced. If the system crashes in the middle of rebalancing, the recovery uses the log to make the tree consistent, then continues to rebalance the tree. The advantage of the incremental logging approach is that only necessary nodes are logged, because it performs undo-logging just before updating the node. If the update doesn't trigger rebalancing, the operation can be performed quickly. However, the tree may be temporarily imbalanced when failure occurs. Furthermore, pcommits and sfences are required for each step.

Given the programming complexity and the frequent persist barriers of incremental logging, we
select the full logging approach for our workloads, where we conservatively log all nodes that may be required for rebalancing on a given operation. Figure 3.5 illustrates the entire root-to-near-leaf nodes are logged prior to inserting node 4. If rebalancing is triggered, no additional logging is required. The advantages of this approach are that it only requires a set of four pcommits regardless of whether rebalancing is triggered or not, and the tree is always balanced regardless of when failure occurs. However, this approach requires logging of additional nodes that may or may not be modified, and this may be expensive if the tree is large.

3.3 Speculative Persistence

3.3.1 Architectural Requirements

As seen in the linked list example, operations on persistent data structures tend to require frequent and clustered sfence, pcommit, and clwb instructions. These instructions in conjunction with sfences incur long pipeline stalls. Based on this observation, we propose to reduce the pipeline stalls by executing speculatively. Rather than letting the sfence stalling the processor while waiting for a pcommit to finish, we propose to checkpoint the architectural state\(^3\) and retire the sfence speculatively. This lets the processor continue executing and retiring instructions speculatively, while the pcommit completes in the background non-speculatively. Note, during speculative execution we must buffer speculative stores and prevent them from updating memory, as in prior speculation schemes [Cez06; Cez07; Ham04; GF02]. Since we speculate on the successful completion of the persist barrier and all stores prior to the barrier, we call our scheme speculative persistency (SP). SP

---

\(^3\) The checkpoint is taken by hardware and stored in registers, and it includes the register file, the PC, and any other register required to restore execution back to that point on a rollback.
allows the overlap of the long latency pcommit operation with the instructions that follow.

SP differs from typical speculations in a few unique ways, necessitating a novel design. First, we use transactions for fail safe persistence, and not for concurrency management. SP does not constraint the use of concurrency management, i.e. it is possible to use SP in conjunction with locks for concurrency management. Second, in SP, speculation failure occurs not due to conflicting accesses, but due to serious system failures (system crash, irrecoverable NVMM errors, etc.). Thus, the probability of speculation failure is very low and rollback can be expected to be extremely rare. This situation may change if we target multi-threaded workloads, depending on what transactions are used for, and the persistency and memory consistency models. However, we note that our study targets sequential programs, and a large class of applications today are still sequential, so it is important to accelerate them. Multi-threaded workload and issues are left for future work. One consequence of the low rollback rate is that rollback and recovery time is much less important than the speed of executing the speculative region. Another consequence is that speculation is less risky, hence we must attempt it as long as there are still opportunities to do so.

The final unique situation for SP is that PMEM instructions (pcommit, clwb, and clflushopt) themselves cannot be executed speculatively, because for them to be considered complete, they have to go to memory. Once they go to memory, they are no longer speculative and can no longer be rolled back. In contrast, regular store instructions can be buffered and considered complete. This is a major problem since, as illustrated in the linked list example, they occur close together and in clusters. Most speculation frameworks simply stop at such instructions. For SP to be effective, we must figure out a new design not to allow them to prevent speculation.

So far we have discussed the unique challenges SP faces, now we will discuss the approach we take in designing SP. First, we need to define the period of speculative execution, from the fence to the last speculative instruction, as a speculative epoch. The speculative epoch begins at an sfence and it ends when the sfence instruction would have otherwise retired from the processor. For example, it would retire when it receives acknowledgement from the memory controller that its buffers have been flushed. At the moment it would have retired, its checkpoint is discarded because it is no longer needed, and its pending memory operations are allowed to complete and update memory. Also, the core returns to a non-speculative mode of execution.

As discussed above, if PMEM instructions are encountered in the speculative epoch, they cannot be executed speculatively. To partially address this problem, we propose that persistent operations be delayed until the end of the speculative epoch, at which time they execute as quickly as possible. Depending on resource constraints and the number of such operations, some serialization may be inevitable.

This choice implies a re-ordering of the PMEM instructions with respect to loads and stores within the epoch. More precisely, given a point in the epoch where the persistent operation occurs, p, and the last instruction that is part of the epoch, end, then the operation is re-ordered with respect
to all memory operations between \( p \) and \( end \). This is architecturally allowed for PMEM instructions.

Both pcommit and clwb have flexible ordering policies, allowing re-ordering with respect to all instructions other than mfence, sfence, xchg, or LOCK-prefixed instructions\(^4\). As long as instructions with strong ordering constraints are not present, the re-ordering is allowed. Hence, we require that epochs only contain PMEM instructions that may be legally delayed until the end of the epoch.

This requirement has an important implication. Clwb, clflushopt, and pcommit are not ordered with respect to loads or stores, so these operations can be moved to the end of a speculative epoch arbitrarily. However, they may not be moved past fences, LOCK-prefixed instructions, or XCHG. Hence, these instructions necessarily form boundaries for speculation, and we must end an epoch at these instructions. Even if we delay PMEM instructions, we cannot extend a speculation past an sfence if it already includes other PMEM instructions.

Given our workload characterization, we know that sfence-pcommit-sfence sequences are likely in the code following the first pcommit. If we are forced to stop speculating at the first sfence, speculation will not proceed far. To overcome this, we propose to use multiple speculative epochs.

Whenever we reach an instruction that cannot be re-ordered with respect to a prior PMEM instruction, we create a new checkpoint and begin a new speculative epoch. Such an epoch will be referred to as a child epoch. For each epoch, the architecture must track its speculative state and PMEM instructions separately.

Furthermore, speculative epochs must commit in sequence from the oldest to the youngest child to ensure that all ordering constraints between speculative epochs are enforced. This additionally implies that all of an epoch's PMEM instructions must commit and complete before any operations from the next speculative epoch can commit.

### 3.3.2 Implementation

Figure 3.6 provides a block diagram of our architecture. To a conventional superscalar processor, we add support for speculation. When a speculative epoch begins, a checkpoint is captured by copying the values of all architectural registers into one of the checkpoints, and the sfence which triggers the checkpoint is removed from the ROB. Speculative stores and PMEM instructions are removed from the ROB at retirement and placed in a Speculative Store Buffer (SSB) until the epoch commits.

Meanwhile, load instructions that follow must check the SSB to satisfy memory dependences with previously retired speculative stores. To ensure loads are handled efficiently, we add a Bloom Filter (BF) to summarize the contents of the SSB and avoid SSB lookups when possible. The Block Lookup Table (BLT) shields the speculative state from becoming visible from other cores.

\(^4\)pcommit is also ordered with respect to serializing instructions, cpuid for example.
3.3.2.1 Speculative Epochs

In general, we can divide the execution of an epoch into three points: the first instruction where it begins, the last instruction of the epoch, and the point in time at which it commits its speculative state.

In our architecture, speculation begins when an sfence following a pcommit is waiting for acknowledgements from the memory controller(s). This would constitute the first epoch in a group of epochs. The first epoch may end either when the pcommit’s acknowledgements are received or in the creation of a child epoch. If the pending pcommit completes before the creation of a child epoch, then the epoch ends and commits at the same time. The checkpoint is discarded and all state in the SSB is allowed to proceed with updating the cache. Meanwhile, the processor resumes non-speculative execution.

However, it is more likely that the first epoch ends by creating a child epoch. In such a case, a child epoch begins on an instruction with a strong ordering requirement, such as a fence instruction, XCHG, or other LOCK-prefixed instructions, since these cannot be re-ordered. The hardware creates the child epoch by taking a new checkpoint using the current speculative architectural state. The child epoch then begins executing.

After the creation of a child epoch, the previous epoch is done executing, but it may not yet be ready to commit. If it is the first epoch, then it must still wait for the pending pcommit to complete. If
it is not the first epoch, then it must wait for its predecessor to complete. Once an epoch’s predecessor fully commits its speculative state, the child can become non-speculative and commit its state as well. This process occurs repeatedly until all child epochs commit their state and the processor resumes non-speculative execution.

In the event that a child epoch is needed but no checkpoints are available, the processor must stall and wait for a checkpoint to become free.

### 3.3.2.2 Speculation using the SSB

To support epoch execution, the SSB will hold speculatively retired stores and PMEM instructions for each epoch. The SSB is a queue and maintains the order of stores and the order of PMEM instructions within an epoch and across epochs. Each entry of the SSB contains an opcode, address, the data if a store, and a checkpoint number to identify which speculative epoch it belongs to. This information enables the SSB to track which epoch the instruction belongs to and when it should commit.

When an epoch commits, the instructions in the SSB update the cache or memory in sequence as quickly as possible depending on the availability of ports to the cache. Keep in mind that speculative epochs may only contain instructions which can be legally re-ordered to the end of the epoch, so re-executing all stores and PMEM instructions at the end is sufficient to safely commit the speculative epoch.

Some instructions and instruction sequences need special support. In particular, we need not preserve fence instructions in the SSB. They are implicitly handled since we require that operations from one epoch must complete before a next epoch can commit. Also, it would be wasteful to devote an entire checkpoint to a single instruction. For example, if a speculative epoch encounters a persist barrier, it will be forced to create a child epoch. Because of the sfence-pcommit-sfence sequence, it would naively create two new checkpoints, one at each sfence with only a single pcommit associated with the first checkpoint. Checkpoints are a limited resource, so we optimize this case for efficiency. Instead of forming two checkpoints, we form a single checkpoint only for the last sfence, and we place a special opcode in the SSB indicating that an ordered pcommit is required before committing the next epoch.

Figure 3.7 illustrates the SSB’s operation over time. When the first pcommit+sfence is encountered, a first epoch starts and a checkpoint is taken. The next two stores (st A and st B) are placed in the SSB and retire. The second pcommit+sfence results in a second checkpoint. When the first pcommit is completed, st A and st B are released to the cache. For added efficiency, we separate the SSB from the store queue and we assume that it is only accessed by loads during speculative execution. However, as long as any stores remain in the SSB, loads must check this buffer for dependences before (or in parallel to) accessing the L1D cache. However, in our analysis, a 256-entry SSB requires 5-cycle to access, longer than a typical L1D cache. To avoid the SSB becoming a perfor-
mance bottleneck, we adopt a bloom filter as in CPR [Gan05]. In our case, the bloom filter is 512 bytes in size. It is set as a store is inserted into the SSB. The bloom filter is reset completely upon exiting speculative execution. Thus, the bloom filter can only produce false positives but not false negatives, and the periodic resets keep the false positive rates low.

![Diagram of Speculative Persistence](image)

**Figure 3.7 Speculative persistence**

While SP does not intend to address concurrency management, and we purposely leave it out of the scope of this work, we need to ensure that SP does not create new persistency problems for a traditional parallel program that were not present without speculation. The scenario that can cause problems is when a core is in a speculative epoch and a coherence intervention from another core occurs. It would be incorrect to reveal speculative state to the other core, and it would be equally incorrect to allow speculation to proceed with incoherent data. Neither option is allowed. Conflicts between external coherence requests and local speculative state must be detected to trigger a rollback. To detect a conflict, as SC++ [Gni99], we added a block lookup table (BLT) that holds a list of all the cache block addresses accessed by both speculative loads and stores. Coherence operations are checked against the BLT; any BLT match is treated as atomicity violation and trigger an abort and rollback. The rollback is to the oldest uncommitted checkpoint. To keep the design simple, currently our BLT design does not distinguish the addresses from different speculative epochs. If a conflict is detected, we simply roll back to the oldest checkpoint, flush the SSB and all speculative state.
3.4 Methodology

3.4.1 Simulation configuration

We implemented clflushopt, clwb, and pcommit instructions in a processor simulator built on MarssX86 [Pat11]. MarssX86 is an open source cycle-accurate full system simulator for an x86-64 architecture. It supports detailed out-of-order CPU, cache, and memory controller models. Table ?? shows the detailed parameters and architecture configurations of the processor and memory system in our simulation. Our machine model includes an out-of-order issue core with three cache levels, backed by an NVMM with 50ns/150ns read/write latency. The NVMM latencies are similar to those assumed by prior work [Yan15; Wan15b; Lee09; Mor13; KK09].

<table>
<thead>
<tr>
<th>Table 3.2 The baseline system configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>L1I and L1D</td>
</tr>
<tr>
<td>L2</td>
</tr>
<tr>
<td>L3</td>
</tr>
<tr>
<td>SSB</td>
</tr>
<tr>
<td>Checkpoint Buffer</td>
</tr>
<tr>
<td>NVMM</td>
</tr>
</tbody>
</table>

In MarssX86, the ordering constraint of clwb is implemented as Intel manual [Int16b], where clwb is ordered only by store-fencing operations (e.g. sfence and mfence) and older stores to the same address while pcommit is only ordered by store-fencing operations. Like regular stores, clwb accesses cache after it is retired in CPU pipeline. The clwb become globally visible, when the dirty cache block is written back to the buffer in memory module. Pcommit is similar except it becomes globally visible after all dirty blocks are persisted to NVMM.

To implement speculative persistence, we implemented a FIFO buffer as a speculative store buffer in between pipeline and cache. Under speculative execution mode, all stores go to speculative buffer and loads access this buffer before cache access. Because memory fence is retired when all previous memory operations are globally visible, the time when clwb and pcommit become globally visible is important. In our implementation, once clwb is issued to memory controller and the acknowledgement from memory controller is replied back to CPU, it becomes globally visible. Likewise, once pcommit is completed and its acknowledgement is replied back to CPU, it becomes globally visible. Even though the speculative persistence is completed, retired stores from pipeline are accumulated to SSB until all stores in the buffer are committed to caches to keep the store order.
Table 3.3 SSB configurations and parameters.

<table>
<thead>
<tr>
<th>Num entries</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (cycles)</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

Once speculative persistence is triggered, loads have to access SSB and check any store to load dependencies before L1D is accessed. Since SSB consists of CAM, RAM, and peripherals, its access latency cannot be ignored. In our experiments, we assumed that CAM and RAM is accessed sequentially, but other peripherals are ignored for simplicity. Table 3.3 shows the access latency on each SSB size. The access latency grows significantly as the size increases.

3.4.2 Benchmarks

Currently, no full applications ported to PMEM are available to us. Thus, we used multiple types of trees in addition to lists, common data structures in various applications. Our benchmarks methodology is borrowed from recent studies [Zha13; Lu14; Cob11; Jos15; Pel14]. The benchmarks that form the workload are shown earlier in Table 3.3. The table shows the number of initial operations that are executed first to populate the data structure, which are fast-forwarded in simulator. For LinkedList, since the search time increases proportionally with the number of nodes, we limit its maximum nodes to 1024, so that the search time does not dominate. For simplicity, we assume that a deleted node is not immediately garbage collected, so that it can be reclaimed if a transaction fails.

For self-balancing tree benchmarks (AVL tree, B tree, and RB tree), we use full logging, which uses four pcommits per operation. We always assume the worst and log all nodes which may be involved if rebalancing is triggered.

3.5 Evaluation

3.5.1 Overall Performance

In order to quantify the various sources of overheads, Figure 3.8 shows execution time overheads for successive additions to each benchmark: adding undo logging code (Log), adding PMEM persistence instructions including clwb, clflushopt, and pcommit (Log+P), and adding fence (Log+P+Sf). While only Log+P+Sf is a correct and fail safe version, Log and Log+P help us debug the performance-robbing factors. The final bars show our SP with 256-entry speculative store buffer (SP256). The baseline is the original benchmarks without any logging or persistence. The final set of bars show the geometric mean of overheads, calculated by geometrically averaging the slowdown ratios, and subtract one from it.

The figure shows that adding logging code already causes significant performance overheads,
26% on average, but higher in tree benchmarks (28% for AVL tree (AT), 95% for B tree (BT), and 83% for RB tree (RT)), due to many more nodes being logged. Non-tree benchmarks only suffer less than 5% overheads from the logging code. Adding PMEM instructions increase the overheads only slightly, to 33% on average, so the PMEM instructions by themselves do not contribute much overheads. However, when the store fences are added, the average overhead shoots up to 60%. Interestingly, non-tree benchmarks, which only log few nodes per operation, such as graph (GH), HashMap (HM), and String Swap (SS), suffer very significant slowdown from the addition of sfences. This points out to the sfences as the primary bottleneck for these benchmarks. Also, considering that logging is necessary for ensuring fail safety, sfences are also the primary bottleneck for tree benchmarks.

Finally, our scheme SP successfully brings down the overheads to 38%, which is only slightly higher than Log+P, indicating that most of the overheads due to the pipeline stalls introduced by sfences, have been removed.

Figure 3.9 shows the number of committed instructions in each benchmark, divided by the original number of committed instructions. It shows that the logging code is the primary contributor in instruction count increase. PMEM instructions only add slightly to the committed instruction count, and sfences count is negligible. This confirms that the slowdown from sfences cannot be due to the increase in the instruction count. Figure 3.10 shows the number of fetch queue stall cycles divided by the original number of execution cycles. It further corroborates that the overheads of sfences come from pipeline stalls: the fetch queue stall cycles of Log+P+SF are much higher than those of Log+P. The figure also shows that these additional fetch queue stall cycles are nearly eliminated with our scheme SP256, bringing them only slightly above Log+P.
3.5.2 Speculative Store Buffer Size

Let us now consider the checkpoint buffer and speculative store buffer design. In designing the checkpoint buffer, we need to figure out how many checkpoints we need to be able to create and keep. To measure that, Figure 3.11 shows the number of maximum concurrent pcommits, which represents the maximum number of pcommits that are encountered while there is at least one older pcommit that has not completed. This data is collected using the Log+P version without the presence of sfences. The figure shows that the maximum number of concurrent pcommits for most benchmarks is four, meaning that a checkpoint buffer with four entries is sufficient.

Next, we use Log+P versions of the benchmarks to count how many store instructions are
executed in the pipeline while a pcommit is outstanding. Figure 3.12 shows the total number of such stores divided by the total number of pcommits. This store count includes clflush and clwb. The figure shows that the number of store instructions is less than 20, except for SS. Taken together, we can infer that the SSB needs to have as many entries as the number of concurrent pcommits multiplied by the number of stores that are executed for each outstanding pcommit. From the two previous figures, we can infer $4 \times 20 = 80$ entries. Note, however, in the Log+P version the pipeline executes faster, so 80 entries for the SSB is likely a floor for a good design.

Figure 3.13 shows the execution time overheads of SP with various number of entries. The figure shows that on average, 256 entries shows the best performance, but we also note 128 entries produces nearly as good performance as 256. The overheads increase as the SSB increases beyond 256 entries due to the high access latency. The overheads increase as the SSB decreases from 128 due to structural hazards of not being able to continue speculative execution.
3.5.3 Bloom filter

Figure 3.13 The overhead of different configurations for speculative store buffer size over baseline.

Figure 3.14 The rates of bloom filter’s false positive.

Due to the overhead of SSB access latency, we use bloom filter to let a load quickly check for a match of address with stores in the SSB. The bloom filter can produce false positives but not false negatives. Figure 3.14 shows the false positive rates of a 512-byte filter. The figure shows that relatively low false positive rates except for SS. We also found that the false positive rates are not caused by the size of the bloom filters, so enlarging it does not help much. Instead, the false positives occur when stores have completed and left the SSB while the bloom filter has not been reset yet.
3.6 Related Work

**Speculative execution.** Speculative execution is a useful and versatile tool in computer architecture design and has been widely studied in a variety of contexts. Researchers have applied speculative execution to sequential consistency to compensate for its performance loss compared to relaxed consistency models [Cez07; Wen07; Blu99; Ran97; Gha91; Gan05; GF02; Pai96; Ham04]. This form of speculation allows aggressive reordering of loads and stores for higher performance. Also, speculation has been proposed for other purposes such as hiding L2 miss latency [Cez06; Mut03; Kir05; Sri04], and speculating past synchronization operations [MT02; RG01; HM93]. It has also been used to ease the burden of parallelization by supporting speculative parallelization [SM98; Cin00; Ste00] and transactional memory [HM93; Ham04; Ana05]. Hardware Transactional Memory is implemented in Intel’s current processors using their TSX extension. Support for multiple checkpoints has been described before in various contexts, like [Cez06; Ham04].

We are the first to apply speculation to memory persistency and identify its primary architectural requirements based on our workloads. Specifically, PMEM instructions that occur in the shadow of a prior pcommit instruction must be delayed and executed at the end of a speculative epoch. This in turn requires supporting multiple checkpoints and speculative epochs to speculate past multiple persist barriers.

**Speculative persistence.** A previous research also used the term *speculative persistence* [Lu14], but our work assumes a different recovery model. The paper allows stores separated by a persist barrier to be persisted in any order but revealed to software in transaction order. This breaks recoverability of our benchmarks because the logging is performed in software in our case. The paper requires a more sophisticated/complex hardware for logging, bookkeeping, and recovery compared to what we propose.
In Chapter 4 we propose Proteus, a software supported hardware logging on NVMM. The remainder of the paper is organized as follows. Section 4.1 provides background on memory persistency, new PMEM instructions, and previous logging implementation for transactions. Section 4.2 introduces software supported hardware logging approach. Section 4.3 describes the detail Proteus designs, Section 4.4 describes the evaluation methodology, and Section 4.5 evaluates our design and presents our key findings. Section 4.6 evaluates different configurations and their impacts on performance. Section 4.7 discusses related work.

4.1 Background

4.1.1 Memory Persistency Models

A persistency model is a specification of the allowable orderings in which stores persist (i.e. are made durable in the NVMM) with respect to the order in which stores appear in the program order. Persistency models give programmers a means to reason about the order of persists, when persists
become durable, and failure-safety. Previous research proposed several memory persistency models, starting from a very high level of abstraction based on durable transactions [Kol16b; Vol11], where all stores in a transaction persist entirely or none of them do. In this model, stores within a transaction are not ordered. Only the ordering of stores in a transaction with stores outside the transaction are enforced.

At a lower level of abstraction, strict persistency, epoch persistency, buffered epoch persistency, and strand persistency, have been proposed [Pel14; Kol16a]. They give programmers a guarantee of the ordering in which stores are persisted to the NVMM, regardless of the presence of durable transactions. Strict persistency [Pel14] piggy-backs on the sequential consistency model by specifying that a store that is globally visible must also have persisted. Due to this constraint, before each store persists to NVMM, all previous stores must have persisted to NVMM. While it is easier to reason about failure safety under this model, it comes with significant performance costs of not allowing write reordering and write coalescing that naturally occur in write back caches. Epoch persistency relaxes the ordering constraint in strict persistency [Con09; Pel14]. It allows the processor to put a persist barrier which defines an epoch in the program. Stores from an epoch (i.e. between two persist barriers) can persist in any order, but they must all persist before the persist barrier. Write coalescing can occur for stores from the same epoch. At the persist barrier, the processor may stall waiting for all stores from the epoch to persist. The buffered epoch persistency model [Con09; Jos15] relaxes the persist barrier by not forcing prior stores to persist right away (hence the processor may not stall), as long as stores from one epoch persist prior to any stores from the next epoch. Strand persistency [Pel14], on the other hand, relaxes the ordering constraints for persists separated by a strand barrier. No ordering is enforced on persists in different strands other than those implied by persist atomicity.

Even lower in the abstraction level is a set of primitives that can be used to specify the ordering of store persistence. Intel PMEM [Int16d] is an example of this approach. PMEM was designed to be compatible with x86 systems, and includes a few new instructions, such as clwb, cflushopt, and pcommit. clwb and cflushopt flush a dirty block from caches to a write pending queue (WPQ) in the memory controller (MC), while pcommit flushes the dirty block from WPQ to the NVMM. By appending either cflushopt or clwb and pcommit after stores that need to persist, programmers can selectively choose which stores should persist and when they persist. cflushopt is ordered with respect to other stores on the same cache block, however clwb is only ordered with respect to stores to the same address. To force an ordering among two of these instructions, a store fence instruction (sfence) is required. sfence, which was originally introduced as a memory barrier to control the visibility of a store with respect to other threads, is extended in PMEM so that it waits for all pending PMEM instructions to complete before retiring. This prevents following stores and PMEM instructions from executing until the sfence completes.

Another important concept, related to the persistency model, is the persistency domain. The
Persistency domain is an architectural description of which components in the system are persistent. Once a store reaches the persistency domain, it can be considered durable. Often, only the NVM itself is in the persistency domain. However, moving the persistency domain on-chip and closer to a core can significantly reduce the time to complete a persist operation. Intel has proposed such an optimization called Asynchronous DRAM Refresh (ADR). While the name refers to DRAM, a key feature of this specification is that data in the WPQ in the memory controller can be considered part of the persistency domain. This change makes pcommit unnecessary since pending operations in the WPQ are already in the persistency domain and do not need to be forced to NVM. Consequently, Intel has deprecated the pcommit instruction.

Due to its lower abstraction level, PMEM instructions can be used to implement some of the other persistency models. For example, if a sequence of (clflushopt, clwb) is inserted after each store, strict persistency is achieved. An example is illustrated in the following code. The first column shows strict persistency where st X, st Y, and st Z are strictly ordered. The second column shows epoch persistency where st X and st Y fall within one epoch, but st Z is in the next epoch.

<table>
<thead>
<tr>
<th>Strict Persistency</th>
<th>Epoch Persistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1: st X, 1;</td>
<td>i8: st X, 1;</td>
</tr>
<tr>
<td>i2: clwb X;</td>
<td>i9: st Y, 1;</td>
</tr>
<tr>
<td>i3: sfence;</td>
<td>i10: clwb X;</td>
</tr>
<tr>
<td>i4: st Y, 1;</td>
<td>i11: clwb Y;</td>
</tr>
<tr>
<td>i5: clwb Y;</td>
<td>i12: sfence;</td>
</tr>
<tr>
<td>i6: sfence;</td>
<td>i13: st Z, 1;</td>
</tr>
<tr>
<td>i7: st Z, 1;</td>
<td></td>
</tr>
</tbody>
</table>

In this paper, we assume durable transactions as our persistency model, where all stores in a transaction either persist together or not at all [Kol16b]. Similar to ADR, we include the memory controller in the persistency domain, which allows our scheme a significant opportunity to reduce the number of writes to NVMM.

4.1.2 Failure Safety

Applications manipulating important data such as database or file-system must ensure that the data is consistent under power failure (failure safety). To construct a durable transaction, where a group of stores are made durable atomically, two approaches are popular: copy-on-write (COW) and write-ahead logging (WAL). With COW, a write triggers data to be copied to a new location where the write will occur. The original data is left intact. COW requires address remapping so that future reads can be redirected to the new location. This remapping is generally considered expensive. For block-based storage, the cost can be amortized by applying copying to a large granularity such as
a page, and committing updates infrequently [Con09]. However, NVMM access is byte-based and stores occur much more frequently than file writes. Thus, COW may be prohibitively expensive to use in NVMM.

With WAL, a redo/undo log for all desired changes is persisted in NVMM prior to committing any modifications. If a failure occurs in the middle of transaction processing, the saved log is used to redo/undo the transaction [Shi17]. Although each update requires two writes, it is not as expensive as an update in COW. WAL can be performed frequently in a small granularity such as the cache block size. The following steps show an example of how undo logging in software can be constructed to ensure failure safety:

- **Step 1** Perform undo-logging and persist undo log.
- **Step 2** Set `logFlag` and persist it, indicating transaction start.
- **Step 3** Update data structure and persist it.
- **Step 4** Unset `logFlag` and persist it, indicating transaction completion.

**Figure 4.1** Steps needed to implement fail-safe undo logging in software.

`logFlag` is used to indicate the progress status of the transaction. If failure occurs before Step 2 completes, the transaction is retried. At re-execution, the log will be overwritten. If failure occurs after Step 2 completes but before Step 4 completes, the log is used to undo the transaction, prior to transaction re-execution. In this implementation, each step must persist before the next step is executed. If PMEM is used, at the end of each step, modifications in the cache must be flushed using `clwb` and `sfence`. These instructions are inserted to prevent reordering with the next step. In this paper, we adopt this implementation as our baseline.

### 4.2 Hardware logging

In the previous section, we have discussed how logging is necessary to support durable transactions that are critical in achieving failure-safe applications. In this section, we introduce our new design, *Proteus*, which performs logging in hardware in a flexible way.

#### 4.2.1 Software Supported Hardware Logging

As discussed earlier, with durable transaction support, programmers can achieve failure safety by grouping related writes into an atomic section. The atomic section requires WAL to ensure a log is created prior to making modifications to data in the NVMM. Log creation, maintenance, and truncation, can be performed in software (*software logging* or SL) or in hardware (*hardware
Figure 4.2 Comparison between software logging and hardware logging.

logging or HL). SL, traditionally a popular technique for persistent (block-based) storage devices, has recently been adapted for use in NVMM, e.g. Mnemosyne [Vol11] and NV-Heap [Cob11]. SL is flexible: it is compatible with a wide range of systems, it does not impose any restrictions on the transaction size or count, and can be changed without impacting the underlying architecture. However, the flexibility also comes with significant costs. First, there are additional instructions that must be executed for creating and maintaining logs, including stores and cache line write-back instructions. Second, a log must persist prior to data modifications, hence the log creation directly lengthens the critical path of transaction execution. Third, when implemented with PMEM, memory fences are needed to impose ordering between the steps shown in Figure 4.1. In our experiments, across a variety of benchmarks, on average SL makes the execution time 1.51x longer.

In response to the execution time and instruction overheads of SL, hardware logging (HL) has recently been proposed, e.g. ATOM [Jos17]. With HL, software indicates the transaction start and end, while a hardware transaction manager creates and manages the log automatically. HL has several advantages over SL. First, no logging memory instructions are needed. Second, since the processor can distinguish logging memory updates from data updates, their ordering can be ensured in hardware at a finer granularity without reliance on memory fences. The effect of this on execution time is illustrated in Figure 4.2. Finally, the awareness of logging updates allows new optimizations. For example, in ATOM, the posted log optimization allows stores to complete before logging updates are made durable by locking the cache block in the MC, thereby preventing subsequent data updates from being persisted until the logging update becomes durable. Source log optimization relegates the creation of log entries to the memory controller instead of the cache controller. This allows log entries to be created with a lower latency. Both of these optimizations are important for reducing the overhead of logging.
In order to get the high performance of HL without losing the flexibility of SL, we propose a third approach: *software supported hardware logging* (SSHL). The key idea behind SSHL is to still rely on software to perform log creation and maintenance, but allow software to inform hardware which operations are performing logging. This removes the need for hard-wiring log creation/maintenance into hardware, thereby preserving SL flexibility. On the other hand, since hardware can now distinguish which operations are logging operations as opposed to regular loads/stores, it can optimize specifically for logging operations differently from regular loads/stores.

We exploit the combined effect of knowing which operations are logging and the fact that the memory controller is in the persistency domain to drop logging operations from the MC once they are no longer needed. When a transaction ends, we can be sure that all of its data updates are durable, either in the NVMM or in the WPQ at the memory controller. Hence, the logging operations that ensure failure safety are no longer needed. Since we can differentiate them from regular stores or necessary persists, we can simply discard them. This not only helps to save power, but also avoids write amplification due to logging updates that may wear out the NVMM.

### 4.2.2 New Logging Instructions

*Proteus* requires software to inform hardware of logging operations. Each logging operation requires two memory addresses: the *log-from* address to hold the address of the original data and the *log-to* address to hold the corresponding address in the log. In order to indicate logging updates to hardware, *Proteus* needs instruction support. We consider two approaches in designing the logging instructions. In the first approach, we consider using a single instruction to do the entire logging operation. However, this requires the log-from and log-to addresses to be specified in a single instruction. This leads to the complexity of possibly handling two page faults in one instruction. Thus, we consider an alternative approach where we rely on two instructions to specify a logging operation. The first instruction reads data from the log-from address. The second flushes it to the log-to address making the log entry durable. The two instructions have the following format:

```
log-load $LR1 M1 Load 32B block from log-from address M1 to log register LR1
log-flush $LR1 M2 flush data from LR1 to the log entry in NVMM at log-to address M2
```

Here, LR is a register that holds a full log entry consisting of the log data, the log-from address, and some other metadata (see Section 4.3). The LR registers are added to support the logging operation. The log-load reads a 32-byte block from address M1 (the log-from address) into the log register LR1 along with the log-from address. The log-flush writes the value in LR1 to the log-to address M2. The log-load completes when data is received at the log register. The log-flush completes when the flush is received at the WPQ in the memory controller and after the memory controller acknowledges receipt.
When a durable transaction is defined by programmers using `tx-begin` and `tx-end`, a compiler expands each store in the transaction into three instructions, `log-load`, `log-flush`, and store. Figure 4.3 shows an example of such transformation. On the left column, the program expresses a durable transaction region with two stores to addresses A and B. On the right column, the compiler replaces each store with a sequence of `log-load`, `log-flush`, and store. LR1 and LR2 are two 40-byte log registers. The number of log registers is selected such that it does not cause a structural hazard in the pipeline. The number of log registers determines the number of stores that can be logged concurrently. LTA is a special register that records the log-to address. In the figure, the auto-increment addressing mode is used to indicate that after a log-flush the LTA is incremented automatically. At the end of a transaction, `tx-end` instruction is inserted by a compiler in order to inform the end of the transaction to the processor. The actions taken by the architecture when executing `tx-end` are discussed in Section 4.3.

In the figure, the write to A is represented by instructions i2, i3, and i4, while the write to B is represented by instructions i5, i6, and i7. Instructions i2 (log-load) and i3 (log-flush) show a read-after-write (RAW) register dependence on LR1. However, instruction i3 (log-flush) and i4 (st) do not show register or memory dependences between them. This allows i4 to be committed from the pipeline and placed in the store buffer. However, an additional ordering must be added here, where i4 cannot be released to the cache until the prior log-flush completes. Enforcing the ordering requires an additional hardware structure, which will be discussed later.

<table>
<thead>
<tr>
<th>Durable transaction</th>
<th>After code generation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tx-begin</code></td>
<td>i1: <code>tx-begin</code></td>
</tr>
<tr>
<td>A = ...</td>
<td>i2: <code>log-load</code> LR1, A</td>
</tr>
<tr>
<td>B = ...</td>
<td>i3: <code>log-flush</code> LR1, (LTA)+</td>
</tr>
<tr>
<td><code>tx-end</code></td>
<td>i4: <code>st</code> A</td>
</tr>
<tr>
<td></td>
<td>i5: <code>log-load</code> LR2, B</td>
</tr>
<tr>
<td></td>
<td>i6: <code>log-flush</code> LR2, (LTA)+</td>
</tr>
<tr>
<td></td>
<td>i7: <code>st</code> B</td>
</tr>
<tr>
<td></td>
<td>i8: <code>tx-end</code></td>
</tr>
</tbody>
</table>

**Figure 4.3** Example code transformation for our SSHL approach.
4.3 Design Details

4.3.1 Log area allocation and log granularity

Prior to using the log, an application needs to create and initialize the log area. One design question is how the log area is accessed. To answer it, we must consider how recovery after a failure will be supported, and in particular, whether the log is accessible through virtual memory or not. Choosing a virtual address (VA) for the log area implies that the application may be responsible for restoring its state. On the other hand, if a physical address (PA) is used, the operating system (OS) must be responsible for restoring the state of all applications. Using PA for logging requires that both data and log pages are fixed and never swapped out. On the other hand, using VA for logging allows data or log pages to be swapped in/out, but this implies that the page table mapping can be restored across crashes, possibly by making the page table persistent or by leveraging relocatable objects [Int16d].

Implementation complexity is important to consider. PA logging incurs significant hardware complexity because the logging operation breaks the virtual address space abstraction of programs, hence the OS and hardware need to be modified accordingly. For example, address space protection and isolation now require separate OS/hardware mechanisms. A fault cannot be handled by a conventional page fault handler. The HL approach is bound to use PA logging in order to avoid the memory controller from having to interact with an application's page table. On the other hand, SL and SSHL can use VA logging, because an application is responsible for the log area allocation and management. Consequently, our Proteus uses VA logging. Specifically, each thread in the application can allocate one log area. The log area is treated as a circular buffer, so that the log can wrap around when space runs out. We assume that the programmer chooses a log area size that is sufficiently large to accommodate the durable transaction size. If the logs overflow the assigned size in a transaction, the processor raises an exception.

Figure 4.4 shows a high level diagram of our Proteus architecture. At the top left, it shows four new registers. log-start and log-end record the start and end address of the log area, respectively. The cur-log register tracks the current free log entry. Finally, txID records the current transaction ID being executed in the core.

The Logging Data Register (LDR) file contains a number of log entries. Each log entry contains a logging data value and metadata, which contains the log-from address for the log entry and the transaction ID. The logging data value has a size that corresponds to the logging granularity, i.e. the number of consecutive bytes that are logged together with a single logging operation. The choice of logging granularity affects performance: if too small, there is little opportunity for coalescing that leads to a high number of log-flushes and writes to NVMM. If too large, data and metadata may not fit in a single cache line, requiring two writes to perform one log-flush and more writes overall to NVMM. Thus, we choose the logging data size to be 32B, leaving the reminder for metadata. Both
4.3.2 Proteus Architecture

Proteus adds hardware structures shown in grey in Figure 4.4. Several 40-byte log registers (LR) are added to the register file (LDR), to keep the log data and log-from address while logging instructions are executing in the processor pipeline. An LR is allocated when a log-load instruction enters the out-of-order (OOO) pipeline, and deallocated when it is no longer needed for detecting register dependences, i.e. when dependent log-flush instructions have committed. Because LRs can be recycled quickly, we found that eight LRs are sufficient.

LogQ is a structure that keeps track of each logging operation. When a log-flush instruction enters the OOO pipeline, an entry is created in the LogQ. It contains the log-from address (location of the original data in NVMM), log-to address (location of the log entry in the log area), and log-data (data value to be flushed to NVMM). The number of entries in the LogQ determines the maximum number of concurrent log-flush operations. A log-flush operation avoids write-allocate in the cache by directly passing the request to the memory controller (MC), avoiding cache pollution. To avoid a cache coherence issue, the log area is marked uncacheable. When the log-flush is received at the MC, the MC sends an acknowledgment to the LogQ and the entry is deallocated from the LogQ. Since an entry is deallocated relatively quickly, we found that increasing LogQ size has diminishing returns and 16 entries in our evaluation is a fair size.

The LogQ has another important function: imposing ordering between a log-flush instruction and a store to the same log-from address. Recall from our discussion in Section 4.2.2 that for correct failure recovery, the log entry must persist prior to the store persist. Therefore, a store to the same log-from address must remain in the StoreQ and not be released to the cache until the preceding log-flush operation is complete. Likewise, log-flushes have to check preceding stores in the StoreQ before they are released to the cache. Thus, when a store retires and before it's committed to the cache, it checks its address against older entries in the LogQ. One corner case, worth mentioning, is that a full LogQ could prevent a log-flush from creating an entry. In this case, to ensure that no stores bypass the log-flush, we stall dispatch if a log-flush fails to find a free LogQ entry. This will ensure we can enforce the required persist ordering later between log-flush and a following store to the same address.

To achieve higher performance, the LogQ allows log entries to flush out-of-order. This must be done carefully to ensure correctness, otherwise it can jeopardize the correctness of recovery. For example, if two log entries in the same transaction have the same log-from address but different data, one must include updates from within the transaction and cannot be used for recovery. Only the first log entry in program order should be used to recover the state. One solution to prevent use of the wrong entry is to guarantee that the log-to address is assigned in program order for all
log entries. In that way, recovery knows to use the earliest log entry and later ones are ignored. To guarantee this invariant, the log-flush determines its log-to address only after all previous log-flushes in program order have resolved their log-to addresses. In spite of the dependency among log-flushes to compute their log-to address, the LogQ can still hide the latency of logging by enabling the concurrent execution of the actual flushes to the MC. This turns out to be an important performance advantage over ATOM since it serializes log entry creation at store retirement.

Moving on, a structure called the Log Lookup Table (LLT) can be seen in the figure. Before explaining the structure, we observe that there is a significant log temporal locality within a transaction.
That is, for our choice of log data size of 32 bytes, it is often the case that there are multiple stores to different bytes/words of the same 32-byte region. If we are not careful, each of these stores will create a sequence of log-load and log-flush, leading to a high number of logging operations and writes to NVMM. As previously mentioned, any logging after the first one to a given log-from address is simply unnecessary overhead.

Eliminating unnecessary logging can be achieved through compiler analysis. However, the presence of aliased pointers make compiler analysis less effective. We prefer to solve it dynamically by adding the Log Lookup Table (LLT). The LLT keeps the last few log-from addresses in a transaction. If there is a new log-flush operation, its log-from address is checked against the LLT. On a match, the log-load and log-flush instructions complete immediately and are not given a log-to address. On an LLT miss, the log-load and log-flush proceed as usual, and the log-from address is added to the LLT replacing an LRU entry, if necessary, from the LLT. The LLT prevents repeated logging operations to the same log data, reducing the memory bandwidth devoted to logging. In addition, the LLT helps to reduce the size of the log area in NVMM, the LogQ, and the LPQ in the MC. For an LLT of 64 entries, the overhead is only 410 bytes.

When a transaction ends, triggered by the tx-end instruction, the LLT is cleared. This prevents the next transaction from finding stale data in the LLT and mistakenly believing it has already logged data. This is one of the primary purposes of tx-end, but there is also another purpose, as described in the next section.

4.3.3 NVMM Log Write Removal

Another hardware structure in Proteus is the Log Pending Queue (LPQ) in the MC. However, before going into that, we will first assume that the MC only has the Write Pending Queue (WPQ) and there is no LPQ.

With the introduction of ADR, the WPQ is now considered non-volatile. Thus, log entries can be considered durable when they are accepted at the WPQ (and before they are written to the NVMM), and acknowledgement of their completion can be sent to the processor at that time. This has the effect of allowing log-flush to complete sooner and stores released to the cache sooner.

We take an even greater advantage from ADR through an additional optimization. Although not in the critical path, writing log entries to the log area in NVMM is expensive due to the added power consumption and reduced write endurance of the NVMM. We note that log entries are no longer needed after a transaction ends (marked by the tx-end instruction) because all data updates are guaranteed to be durable, either in the WPQ or in the NVMM. In the common case, logs are written once and never read again. Hence, log entries that have not yet been written to NVMM after the transaction ends can be flash cleared and never written. This leads to the insight that we should keep log entries in the WPQ until a transaction ends, if possible, to avoid ever writing them again.

65
to NVMM. This not only helps to save power consumption, but also avoids premature wear-out in the NVMM by significantly reducing the number of writes.

To achieve this, we must prioritize writing back regular writes from the WPQ to the NVMM. This requires a priority bit to be added (or expanded if it is already there) so that log flushes are de-prioritized and rarely released to the NVMM.

Inevitably, some log entries may have been released to the NVMM before the end of the transaction anyway. These log entries need to be invalidated by reading and marking these log entries invalid in the NVMM. As a result, the potential savings are not as great as we might expect. To overcome this issue, ATOM introduces hardware in the MC to track all active log entries and clear them once a transaction is completed [Jos17]. Because of this hardware design, ATOM’s performance benefits are limited to its available resources. Once the resources run out, ATOM has to search the log area and invalidates them manually one by one. We address this issue in Proteus using a simple design. First, we add a transaction ID to the meta data of each logging operation so that all log entries belonging to a transaction can be identified quickly. Second, we allocate a separate log area for each thread. Intuitively, since there is only one active transaction at any particular time within a thread, only the log entries belonging to the most recent transaction are the valid logs.

In the above technique, the log area still needs to mark the end of a transaction (tx-end) to indicate whether the most recent log in the log area is valid or not. Instead of using one more log entry to mark the end of a transaction, Proteus utilizes the meta data of the last log entry for marking the end of the transaction. The last log entry in a transaction still needs to be flushed to NVMM. However, this mark is only necessary before the next transaction from the same thread starts in the log area. So we add a minor optimization that the last log entry is held in the WPQ (except when WPQ is full) and is discarded once a log entry from the next transaction reaches the WPQ.

So far we have described our flash clearing and priority bit optimizations. However, we notice that keeping log entries only in the WPQ has a limitation that logging operations and regular writes compete for entries in the WPQ. Increasing the number of WPQ entries can alleviate this, but it adversely increases read latency. An incoming read must be checked against WPQ entries for a match. A larger WPQ increases the checking time and directly increases the critical path of read requests.

To avoid extending the critical path of reads, we add the LPQ, and log flushes go only to the LPQ freeing the WPQ only for regular write-backs. An incoming read does not check against the LPQ because logs are not used again by the processor except during failure recovery. This also eliminates the additional priority bit in the WPQ. Prioritizing write requests in the WPQ over logging requests in the LPQ can be achieved at the arbiter instead. The LPQ contains log entries, where each entry contains the transaction ID, core ID, and various information about the log. When a transaction ends, all LPQ entries matching the particular transaction ID are cleared.
4.3.4 Context Switch

We also consider the case of supporting a context switch with our new hardware. The new registers added to the core need to be saved, namely the txID, log-start, log-end, cur-end, and LR registers. Also, we need to clear the LLT, just like on a tx-end, to ensure that entries in the LLT are not mistakenly used by another thread. Note, even if the same thread is rescheduled, it may simply result in additional log entries for the same data. However, this can be handled easily during recovery by recovering from the earlier entries in the log.

The other case we need to handle is ensuring that data in the MC is flushed to NVMM. Since we do not know how long the thread may be switched out, we send a message to the MC informing it to write all LPQ entries for the txID to NVMM. This conservatively assures correctness. Since context switches are rare, the performance penalty is minor.

For processors do not have an explicit context switch instruction that can perform these actions, we add the log-save instruction to carry out the actions we described.

4.4 Methodology

4.4.1 Simulation configuration

For our experiments, we implemented Intel PMEM instructions, clwb, clflushopt, and pcommit in a processor simulator built on MarssX86 [Pat11], which is an open source cycle-accurate full system simulator for an x86-64 architecture. We also add the appropriate ordering constraints for memory fences (sfence and mfence) with respect to PMEM instructions. Of the PMEM instructions, we only use clwb because pcommit has been deprecated while clflushopt is not needed (it invalidates a block after flushing it). Our simulation model supports a detailed out-of-order multicore CPU, coherent caches, interconnection, and memory controller models. To simulate a detailed memory system, we also integrated DRAMsim2 into MarssX86. In our implementation, the ordering constraint of clwb and pcommit is implemented as described in Intel’s manual [Int16b], where clwb is ordered with respect to older stores to the same address and store-fencing operations (e.g. sfence and mfence). After clwb is retired from the CPU pipeline, it accesses the cache in the same way as regular stores. If the clwb hits a dirty block in the cache hierarchy, the dirty cache block is flushed to the WPQ in the MC. Once the cache block is placed in the WPQ, the clwb becomes globally visible.

In addition to implementing Proteus, we also implement the state-of-the-art in hardware logging, ATOM [Jos17], for comparison. In ATOM, a log entry is automatically generated right before a store gets retired. Logging delays the store’s retirement and the store is held in the storeQ until the logging operation is completed. In order to compare with the best-performing version of ATOM, we implemented and integrated both posted log optimization and source log optimization into ATOM. Thus, logging is considered complete once the log entry arrives at the MC which locks the cache line.
and sends an acknowledgment back to the cache controller. This optimization reduces the latency of logging, which is on the critical path of the store operation. Furthermore, on a cache miss with a logging operation, a log entry is created in the MC before the data is sent to the cache.

Table 4.1 The baseline system configuration.

<table>
<thead>
<tr>
<th>Processor</th>
<th>OOO, 3.4GHz, 4 cores, 5-wide issue/retire. ROB: 224, fetchQ/issueQ/LoadQ/StoreQ: 48/64/72/56</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 and L1D</td>
<td>32KB, 8-way, 64B block, 4 cycles, private per core</td>
</tr>
<tr>
<td>L2</td>
<td>256KB, 8-way, 64B block, 12 cycles, private per core</td>
</tr>
<tr>
<td>L3</td>
<td>8MB, 16-way, 64B block, 42 cycles, shared by all cores</td>
</tr>
<tr>
<td>Interconnect Bandwidth</td>
<td>96B/cycle for CPU-L1, 64B/cycle for L1-L2 32B/cycle for L2-L3, 16B/cycle for L3-MC</td>
</tr>
<tr>
<td>DRAM</td>
<td>DDR3-1600 (800MHz), 8GB 1 channel, 16 Banks per rank, 2KB row-buffer</td>
</tr>
<tr>
<td>NVM</td>
<td>tRCD 29 for Read, 109 for Write</td>
</tr>
<tr>
<td>Proteus</td>
<td>LR: 8 registers, LogQ: 16 entries, LLT: 64 entries (8way), LPQ: 256 entries</td>
</tr>
</tbody>
</table>

The simulation parameters and architecture configurations for the processor and memory systems are listed in Table 4.1. Our machine model includes a quad-core processor with each core supporting out-of-order issue and execution with three levels of cache backed by DRAM/NVMM. The configuration parameters are similar to Intel’s Skylake architecture [Int16c] with minor differences. The DRAMsim2 parameters are also listed in the table. To simulate NVMM latencies for the baseline memory which assumes DDR3-1600, we increased tRCD to 29 cycles for read and 109 cycles for write (50ns for read and 150ns for write), in line with numbers assumed in prior work [Yan15; Wan15b; Lee09; Mor13; KK09; Shi17; Awa16].

4.4.2 Workloads

Using the PMEM instructions, we constructed a workload consisting of benchmarks with data structures listed in the Table 4.2. The benchmarks are borrowed from or are similar to those used in previous studies [Zha13; Lu14; Cob11; Jos15; Pel14; Jos17; Shi17]. For each benchmark, we construct
an operation that is either a node insertion or deletion (except for String Swap). The operation is wrapped inside a durable transaction. Each benchmark receives an operation type and a key for each operation from an input file which contains the list of operations generated randomly. We used multiple data structures per benchmark to avoid excessive lock contention among multiple threads. In this work, we assumed that multicore shared memory accesses among transactions are solved by thread synchronization using locks, which guarantees mutual exclusion between concurrent transactions. We believe that this is a separate problem and not the focus of our work. And hence, each operation must obtain a lock for a data structure before the operation is performed and no other threads can interrupt the executing thread in the middle of the update. The table shows the number of initial operations per thread that are executed first to populate the data structure, which are then fast-forwarded in the simulator. To eliminate non-determinism from our experiments, the pthread barrier is used after initial operations so that all threads run together as they are simulated.

Table 4.2 Benchmarks constructed for our study. Except for SS which has 256 bytes for each string, we size each node to be 64 bytes and align them to cache blocks in all benchmarks. Thus, to persist one node update, one clwb will be required. InitOps and SimOps are expressed in terms of the number of operations that are executed per thread.

<table>
<thead>
<tr>
<th>Benchmark (Abbrev.)</th>
<th>Description</th>
<th>#InitOps</th>
<th>#SimOps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue (QE)</td>
<td>Enqueue/dequeue in 8 queues</td>
<td>20000</td>
<td>50000</td>
</tr>
<tr>
<td>HashMap (HM)</td>
<td>Insert or delete entries in 16 hash maps</td>
<td>100000</td>
<td>20000</td>
</tr>
<tr>
<td>String Swap (SS)</td>
<td>Swap strings in a string array (262144 items)</td>
<td>20000</td>
<td>50000</td>
</tr>
<tr>
<td>AVL tree (AT)</td>
<td>Insert or delete nodes in 16 AVL trees</td>
<td>100000</td>
<td>10000</td>
</tr>
<tr>
<td>B tree (BT)</td>
<td>Insert or delete nodes in 16 B trees</td>
<td>100000</td>
<td>10000</td>
</tr>
<tr>
<td>RB tree (RT)</td>
<td>Insert or delete nodes in 16 RB trees</td>
<td>100000</td>
<td>10000</td>
</tr>
</tbody>
</table>

On each benchmark, we created a manual undo-logging version and a version for ATOM and Proteus. We found that rebalancing operations in self-balancing tree benchmarks (AVL tree, B tree, and RB tree) are challenging for the creation of undo logs because it is difficult to know which nodes will be modified at the start of the transaction. Therefore, our manual undo-logging assumes the worst and logs all nodes that could be modified by the operation. Furthermore, for simplicity, we assume that memory allocations and deallocations are performed in a failure-safe way so that our undo-logging need not cover them.
4.5 Evaluation

In order to assess the performance benefits of Proteus, we implemented and compared the following schemes: software logging represented by an Intel PMEM based implementation of WAL, both with pcommit (PMEM+pcommit) and without it (as the base case), hardware logging represented by ATOM [Jos17] including all of its optimizations (ATOM), and software-supported hardware logging represented by our scheme Proteus (Proteus) and without log write removal (Proteus+NoLWR). In order to see how close they perform to an ideal case, we also implemented PMEM but with logging removed (PMEM+nolog). The latter does not provide failure safety and is devoid of any logging overheads, and thus it is an ideal case.

The result of their speedup over the base case of PMEM without pcommit for all benchmarks and for the geometric mean of all benchmarks are shown in Figure 4.5. First, let us observe the PMEM+pcommit bars. They are significantly below 1.0 in all benchmarks, with a geometric mean of 0.79. This shows that moving the MC and WPQ into the persistency domain is very helpful for performance. Next, consider the last bars (PMEM+nolog) that are significantly higher than 1.0, with a geometric mean of 1.51. This shows that the addition of logging code and its execution causes very significant performance overheads, whereas its removal speeds up execution by 51% on average. However, on benchmarks with complex data structures such as BT and RT, the speedups when logging is removed are very high: in the case of BT, logging code removal results in a $2.98 \times$ speedup. This is because, for complex data structures, it is difficult to determine which components of the data structure will need to be undo logged. For example, tree balancing operations may affect only a few nodes in the best case or the entire tree in the worst case. Thus, logging code needs to assume conservatively that a high number of nodes will be affected by a transaction.

Now let us examine ATOM and Proteus. ATOM performs quite well, achieving a $1.33 \times$ speedup
on average while Proteus achieves a geometric average of 1.46× speedup. In other words, Proteus is faster than ATOM by \( \frac{1.46}{1.33} - 1 = 10\% \). Furthermore, Proteus's speedup is only 3.3% lower than the ideal case of no logging. In ATOM, a log entry is automatically generated right before a store gets retired and the retirement is delayed until the log entry becomes durable. Due to this constraint, the rate at which store operations are completed is reduced. For ATOM's design, this is necessary to solve the dependency between log entries and stores. However, Proteus does not have this limitation because the LogQ manages the dependency. This allows Proteus to support concurrent logging as long as log-flushes do not have dependences with preceding stores. In addition, it allows stores to complete earlier. We found that concurrent logging provides an important advantage in overall performance.

To analyze the performance difference between ATOM and Proteus, we investigated the stall cycles at the front-end of the pipeline before instruction dispatch. The front-end could be stalled by a lack of free resources in the ROB, physical registers, or LSQ. Figure 4.6 shows the stall cycles normalized to the stall cycles of PMEM+nolog in the front-end. ATOM has 12% more stalls than Proteus and 16% more stalls than the ideal case. On the other hand, the number of stall cycles in Proteus is fairly close to the ideal case, only 4% more stalls. These results show that ATOM creates more pressure on the pipeline and eventually stalls it, but Proteus is free from this limitation.

Figure 4.7 compares the number of NVMM writes for each benchmark, normalized to the number of NVMM writes of PMEM+nolog. On average, ATOM has three times more writes to NVMM (3.4×), compared to PMEM without logging. In benchmark (QE), it more than quadruples the writes to NVMM and in the worst case (AT), it has six times more writes to NVMM. The increase in number of writes is due to logging (creation and truncation). This is significant because it cuts the write endurance of NVMM by more than three quarters. In contrast, Proteus only increases the number of writes slightly. In the worst case (AT), the increase in writes is still relatively low, at 6%. The reason for Proteus's advantage is that most log updates are held at the LPQ and flash cleared when a transaction
ends, thanks to the fact that the MC is part of the persistency domain. Thus, most log flushes do not even go to the NVMM.

![Figure 4.7](image.png)

Figure 4.7 The number of NVMM writes, normalized to PMEM with no logging.

### 4.6 Sensitivity Study

In order to quantify the impact of memory latency on the performance of the logging schemes we studied, we ran our experiments with slower NVMM and faster DRAM. Moreover, later in this section, we expand our study into the impact of the new components of *Proteus* on performance with varied hardware structures and their tradeoffs.

#### 4.6.1 Performance on slow NVM

Current NVM read and write latencies, for various technologies, have not yet reached our previous assumption of fast NVM with 50ns read and 150ns write latencies. In order to see the impact of our scheme on slower NVM devices, we modeled a higher write latency at 300ns while keeping the read latency at 50ns. Not surprisingly, the overall performance of all test cases decreased 10-23% with slow NVMM compared to faster NVMM, indicating that slow write latency affects the performance.

Figure 4.8 shows the speedup of each benchmark on a slow NVMM, with the baseline of PMEM as before. The geometric mean of speedups are 1.33 for ATOM, 1.49 for *Proteus*, and 1.53 in the ideal case. Compared to Figure 4.5, the speedup of the ideal case is slightly improved since it has fewer writes than the baseline. On the other hand, *Proteus* is also less affected by write latency and still maintains superior performance close to the ideal case. However, the speedup of ATOM stays the same over the baseline, indicating that ATOM is more influenced by write latency than *Proteus*. 
On average, **Proteus** experienced only a 10% performance decrease compared to the faster NVMM, which is superior than the 12% decrease of **ATOM**'s. Hence, **Proteus**'s advantage becomes more substantial with longer NVMM write latencies.

### 4.6.2 Performance on DRAM

Figure 4.9 shows the speedup comparison on DRAM. The rationale behind running with DRAM is to study the performance of logging when battery backed DRAM solutions, like NVDIMM, are used. On DRAM, we found that **Proteus** still performs quite well. We observe average speedups of 1.31 for **ATOM**, 1.47 for **Proteus**, and 1.52 for the ideal case. Compared to Figure 4.5, overall
performance is improved from 11% for ATOM up to 30% for PMEM+pcommit. Because of its modest performance improvement, ATOM’s speedup over the baseline decreased slightly. Considering the fact that logging is completed at the MC with posted log optimization, the speedup degradation in ATOM over the baseline is the result of the tradeoff between faster accesses to DRAM and relatively fixed logging latency in ATOM. On DRAM, the overhead of logging plays a slightly bigger role in the overall performance of ATOM. On the other hand, Proteus maintains its superior performance with a 13% performance improvement over NVMM. We found the reason is that it hides the logging latency more effectively using concurrent logging. Although it is not described in the figure, the speedup difference obtained when changing from the LogQ size of 8 to 16 increases notably, from 0.02 with NVMM to 0.11 with DRAM. The bigger LogQ helps Proteus hide the logging latency even with fast memories.

4.6.3 Analysis of logging components

![Figure 4.10](image)

Figure 4.10 Speedup comparison with varying LogQ sizes. The baseline is software logging with PMEM.

Figure 4.10 shows the impact on speedup by varying the LogQ size for each benchmark. From the figure, average speedup shows an increasing trend as LogQ size is varied from 1 to 64, but we also can see the diminishing returns as the size increases. These results suggest that a LogQ size of 8 gives a 1.44x speedup. Speedups start to saturate with a LogQ size of 8, with very little improvement (1-2%) as the size is doubled. A LogQ size of 64 gives more than 1.47x speedup which is only 2.4% below the ideal case. We chose a LogQ size of 16 as our optimal configuration instead of 8 considering its superior performance on DRAM and its dominant performance over ATOM across all benchmarks, particularly since it performs worse than ATOM in benchmark (SS) with only 8 entries in the LogQ.

Figure 4.11 shows the combined interaction of varying LogQ along with LPQ. From the previous
study we pick the optimal LogQ size of 16 and use it to study different LPQ sizes. We found that benchmarks require a certain LPQ size depending on their transaction size. As long as a large enough LPQ is provided, overall performance is unaffected. Otherwise, the performance drops rapidly for smaller sizes. We select the size of 256 entries as our configuration.

Table 4.3 Speedups for large transactions.

<table>
<thead>
<tr>
<th>Transaction Size</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proteus</td>
<td>1.20</td>
<td>1.24</td>
<td>1.23</td>
<td>1.24</td>
</tr>
<tr>
<td>PMEM+nolog(ideal)</td>
<td>1.23</td>
<td>1.25</td>
<td>1.25</td>
<td>1.27</td>
</tr>
</tbody>
</table>

We implemented a microbenchmark with variable-sized, large transactions based on the linked list benchmark. The number of elements updated per node is taken as a variable to stress Proteus. Each transaction completes once all elements on a node are updated, and we chose the number of elements for our experiments as 1024, 2048, 4096, and 8192. We found that the benchmarks generate $20\times$, $39\times$, $78\times$, and $156\times$ more log entries per transaction compared to the existing benchmarks. Although a large number of those log entries are filtered by LLT, we still found that $7\times$, $13\times$, $26\times$, and $52\times$ more log entries per transaction are flushed to the MC. Table 4.3 compares the speedups of Proteus and the ideal case (PMEM+nolog) with varying transaction sizes. The data suggests that the performance of Proteus is still very close to the ideal case. The result also suggests that hardware structures (LogQ, LLT, and LPQ) used in Proteus are able to sustain large transactions.

Table 4.4 shows the LLT miss rate for the benchmarks collected with an LLT size of 64 entries. The data shows that the benchmarks chosen did exhibit varied miss rates from 22.5% to 51.6%. Higher
Table 4.4 LLT miss rate (%) for different benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AT</th>
<th>BT</th>
<th>HM</th>
<th>RT</th>
<th>SS</th>
<th>QE</th>
</tr>
</thead>
<tbody>
<tr>
<td>miss rate</td>
<td>37.2</td>
<td>36.1</td>
<td>39.2</td>
<td>51.6</td>
<td>24.5</td>
<td>22.5</td>
</tr>
</tbody>
</table>

LLT miss rates indicate more log entries per transaction. The data shows that the LLT efficiently absorbs half to three quarters of logging traffic, helping to reduce the memory bandwidth devoted to logging. Furthermore, the decreased log entries help to reduce the necessary size of the LogQ, the LPQ, and the log area in NVMM.

### 4.7 Related Work

The emergence of new non-volatile memory technologies has paved the way for designing systems with fast non-volatile main memory, which has opened many new research directions. Modern systems are optimized for volatile memory, such as DRAM, which makes it difficult to maintain data consistency in NVMM after power failure. In order to maintain data consistency in NVMM, researchers have borrowed traditional storage techniques such as copy-on-write or write-ahead logging and applied them in the context of NVMM. Condit et al. [Con09] proposed a new file system for NVM and used shadow paging, one form of the copy-on-write technique, to provide an atomic file-system update. This paper also proposed epoch persistency to ensure ordering persists. Ren et al. [Ren15] proposed a hardware supported failure-safe system using copy-on-write.

Using a copy-on-write approach for data consistency on NVMM is not easy to implement because remapping in hardware is expensive. Moreover, a large update granularity, which often amortizes the remapping overhead, is less suitable for use in byte-addressable NVMM. Hence, instead of using copy-on-write, the majority of papers on NVMM have adopted WAL which allows transactions at smaller granularity and does not require remapping. For example, Mnemosyne [Vol11] proposed a library to access NVMM directly and used redo-logging to provide an atomic durable transaction. On the other hand, NV-Heaps [Cob11] used software undo-logging for its atomic transaction implementation and improves its performance by logging at a large granularity instead of logging before every write. Rewind [Cha15] proposed a library to manage NVMM directly from the user application and it also utilized a software undo-log approach for its durable transaction.

In general, logging operations are expensive because they force an ordering between logging operations and the following stores. Therefore, recent works discussed about the cost of logging in the transaction and endeavored to reduce it. For example, Atlas [Cha14a] uses software undo-logging in its durable transaction and proposes programming models to optimize them. Pelley et al. [Pel13] proposes NVRAM group commit which groups the transactions executed into a batch and commits them at once, amortizing its commit overhead. Similarly, Kolli et al. [Kol16b] proposed DCT, which
defers a transaction commit on a multi-threaded application and commits them together while resolving possible conflicts among them. Recently, Liu et al. [Liu17] proposed a software logging approach, DudeTM, which efficiently eliminates the ordering constraint using asynchronous redo-logging. However, it creates more memory writes and requires additional memory space from both DRAM and NVMM compared to previous approaches.

Hardware logging approaches for mitigating logging overheads were proposed in multiple papers. Lu et al. [Lu14] proposed LOC which relaxes the ordering constraint in transactions by using asynchronous redo-logging with background hardware supports. Similar to DudeTM, it requires additional memory accesses and space. Furthermore, it requires redirecting reads to logs or blocking reads during the logging operation until it has been committed to memory, which is generally considered an expensive operation. It also needs additional hardware support for updating memory in the background. Doshi et al. [Dos16] proposed synchronous hardware redo-logging for atomic durability. However, their solution is not optimized for the case with multiple updates to the same address in one transaction. In this case, they simply create multiple log entries for the same data, which consumes unnecessary memory bandwidth and also degrades performance. Recently, Joshi et al. [Jos17] proposed ATOM which uses synchronous undo logging and includes hardware to help create only one log entry per update per transaction. They also showed better performance compared to a previous redo-logging scheme [Dos16]. In our work, we show that we attain better performance than ATOM, thanks to concurrent logging and log write removal.

A persistent cache hierarchy has also been proposed and used in multiple works [NH12; Zha13; WJ14; Naw15a; Naw15b; Izr16]. These works bring the data caches into the persistency domain. Hence, stores are considered durable once they leave the store buffer. Obviously, these systems do not have flushing overheads because data does not need to be forced out of the cache, but they still need to manage logging schemes. The ordering constraint between log entries and subsequent stores can be handled in the processor.

A few works [Hua14; Kim16; Oh15] have used NVMM with write-ahead logging for a database system usually placed on disks. NV-Logging [Hua14] shows that replacing a whole disk with NVM is an expensive solution per dollar. Instead, they show it is cost effective to place only a logging subsystem on NVMM enabling concurrent logging for multiple transactions. Similarly, Oh et al. [Oh15] proposes PPL which deploys a similar technique but on a smaller scale with SQLite on mobile systems. NV-WAL [Kim16] optimizes PPL considering byte-addressability, write reordering, and user-level heap management on NVMM.
In Chapter 5 we propose a new page table walk scheduling policy, SIMT-aware scheduler. The remainder of this chapter is organized as follows. Section 5.1 provides background on GPUs and virtual address translations on GPUs. Section 5.2 shows the motivation of this work and the overview of SIMT-aware scheduler. Section 5.3 describes the detail implementation of SIMT-aware scheduling. Section 5.4 describes the evaluation methodology and results from the experiments. Section 5.5 shares further discussion on the page table walk scheduler. Section 5.6 discusses previous related works.

5.1 Background

This work builds upon two aspects of a GPU’s execution: the GPU’s Single-Instruction-Multiple-Thread (SIMT) execution hierarchy, and the GPU’s virtual-to-physical address translation mechanism.

5.1.1 Execution Hierarchy in a GPU

GPUs are designed for massive data-parallel processing that concurrently operates on hundreds to thousands of data elements. To keep this massive parallelism tractable, a GPU’s hardware resources
are organized in a hierarchy. The top of Figure 5.1 depicts the architecture of a typical GPU.

Compute Units (CUs) are the basic computational blocks of a GPU, and typically there are 8 to 64 CUs in a GPU. Each CU includes multiple Single-Instruction-Multiple-Data (SIMD) units, each of which has multiple lanes of execution (e.g., 16). A SIMD unit executes a single instruction across all its lanes in parallel, but each lane operates on a different data item. A GPU’s memory resources are also arranged in a hierarchy. Each CU has a private L1 data cache and a scratchpad that are shared across the SIMD units only within the CU. When several data elements accessed by a SIMD instruction reside in the same cache line, a hardware coalescer combines these requests into single cache access to gain efficiency. Finally, L1 caches are followed by an L2 cache that is shared across all CUs in a GPU.

Figure 5.1 Baseline system architecture.
GPGPU programming languages, such as OpenCL [Gro14] and CUDA [Cud], expose to the programmer a hierarchy of execution groups that follows the hierarchy in the hardware resources. A workitem is akin to a CPU thread and is the smallest execution entity that runs on a single lane of a SIMD unit. A group of workitems, typically 32 to 64, forms a wavefront and is the smallest hardware-scheduled unit of work. All workitems in a wavefront execute the same SIMD instruction in a lockstep fashion but can operate on different data elements. An instruction completes execution only when all workitems in that wavefront finish processing their respective data elements. The next level in the hierarchy is the programmer-visible workgroup that typically comprises tens of wavefronts. Finally, work on a GPU is dispatched at the granularity of a kernel, comprised of several workgroups.

5.2 The Need for Smarter Scheduling of Page Table Walks

Irregular GPU applications often make data-dependent memory accesses with little spatial locality [Cha14b; Bur12]. This causes memory access divergence in the GPU’s SIMT execution model where different workitems within a wavefront access data on distinct pages. The hardware coalescer is ineffective in such cases as several different address translation requests are generated by the execution of a single SIMD memory instruction. These requests then look up TLBs but often miss there owing to less locality in irregular applications. Eventually, many of these requests queue up in the IOMMU buffer to be serviced by a page table walker.

A recent study on commercial GPU hardware demonstrated that such divergent access can slowdown irregular GPU applications by up to $3.7 \times 4 \times$ due to address translation overheads [Ves16]. In this work, we aim to reduce address translation overheads for such irregular GPU applications.

We discover that the order in which page table walks are serviced can significantly impact the address translation overheads experienced by an irregular GPU application. While better page table walk scheduling (ordering) can potentially improve performance, poor scheduling (e.g., random scheduling) can be similarly detrimental. Figure 5.2 shows the extent by which scheduling of page table walks can impact performance on a set of representative irregular applications (methodology is detailed in Section 5.4.1). The figure shows speedups of each application while employing naive random scheduler\(^1\), the baseline FCFS, and the proposed SIMT-aware page walk scheduler. Each bar in the cluster shows the speedup of an application with a given scheduler, normalized to that with random scheduler. While we will detail our SIMT-aware scheduling over the next two sections, the key message conveyed by the figure is that the performance of an application can differ by more than $2.1 \times$ due to the difference in the schedule of page table walks. This underscores the importance of exploring the scheduling of a GPU’s page walk requests.

\(^1\)As its name suggests, the random policy randomly picks a pending page walk request to service from the IOMMU buffer.
A keen reader will notice the parallel between the scheduling of page table walks and the scheduling of memory (DRAM) accesses at the memory controller [Rix00; Kim10a; Aus12; LA16]. The existence of a rich body of research on memory controller scheduling suggests that there exist opportunities for follow-on work to explore different flavors of page walk scheduling for both performance and QoS.

In the rest of this section, we discuss why page table walk scheduling affects performance and then provide empirical analysis to motivate better scheduling of GPU page walks.

5.2.1 Shortest-job-first Scheduling of Page Table Walks

We observe that instructions issued by a wavefront require different amounts of work to service their address translation needs. There are two primary reasons for this. First, the number of page table walks generated due to the execution of a single SIMD memory instruction can vary widely based on how many distinct pages the instruction accesses and the TLB hits/misses it generates. In the best case, all workitems in a wavefront access data on the same page and the perfectly coalesced translation request hits in the TLB. No page walks are necessary in that case. At the other extreme, a completely divergent SIMD instruction can generate page table walk requests equal to the number of workitems in the wavefront (here, 64). Second, each page walk may itself need anywhere between one to four memory requests to complete. This happens due to hits/misses in page walk caches (PWCs) that store recently-used upper-level entries of four-level page tables (detailed in Section 5.1).

Figure 5.3 shows the distribution of the number of memory accesses required to service address translation needs of SIMD instructions for a few representative applications. The x-axis shows
buckets for the number of memory accesses needed by a SIMD memory instruction to service its address translation needs. The y-axis shows the fraction of instructions issued by the application that fall into the corresponding x-axis buckets. We excluded instructions that did not request any page table walks. We find that often between 27-61% of the instructions needed one to sixteen memory accesses to complete all the page table walks it generated. On the other hand, more than 33-70% of the instructions required forty-nine or more memory accesses. One of the applications (GEV) had close to 31% of instructions requiring sixty-five or more memory accesses. In summary, we observe that the amount of work (quantified by the number of memory accesses) required to service the address translation needs of an instruction varies significantly.

![Figure 5.3 Distribution of number of memory accesses (i.e.,‘work’) for servicing address translation needs of SIMD instructions.](image)

It is well studied in scheduling policies across various fields that in the presence of “jobs” of different lengths, if a longer job can delay a shorter job, then it impedes overall progress. This leads to the widely employed “shortest-job-first” (SJF) policy that prioritizes shorter jobs over longer ones [SJf]. By analogy, we posit that servicing all page table walks generated due to the execution of a single instruction should be treated as a single “job” because the instruction cannot complete until all those walks are serviced. Figure 5.3 demonstrates that the “length” of such jobs, as quantified by the number of memory accesses, vary significantly.

**Key idea (1):** Following the wisdom of time-tested SJF policies, we propose to prioritize the servicing of page table walk requests from instructions requiring fewer memory requests to complete their address translation needs over those requiring larger number of memory accesses.
5.2.2 Batch-scheduling of Page Table Walk Requests

Owing to the GPU’s SIMT execution model, all page table walks generated by a single SIMD instruction must complete before the instruction can finish execution. The performance is thus determined by when the last of those walk requests is serviced. Even servicing all but one walk request does not aid progress.

Figure 5.4a illustrates how the progress of two SIMD instructions, `load A` and `load B`, issued by two wavefronts are impaired if their page table walk requests are interleaved. Both `load A` and `load B` generate multiple walk requests and both experience stalls due to the latency to service walk requests generated by the other. Evidently, if interleaved page walk requests are serviced in the FCFS order, then it delays completion of both `load A` and `load B` since both need all their walk requests to finish before the instruction can progress. This inefficiency exacerbates if walk requests from a larger number of distinct instructions interleave since the progress of every instruction involved in the interleaving suffer.

Unfortunately, such interleaving among page walk requests from different SIMD instructions happens fairly regularly. Figure 5.5 quantifies how often such interleaving happens for representative irregular GPU workloads (methodology detailed in Section 5.4.1). The y-axis shows the fraction of executed memory instructions whose page walk requests interleave with requests from at least another instruction. We exclude any instructions that do not generate at least two page table walks as interleaving is impossible for them. We observe that 45-77% of such instructions have their walk requests interleaved.

We traced the source of this interleaving to the GPU’s shared L2 TLB. The shared L2 TLB receives
multiple independent streams of address translation requests generated by L1 TLB misses from concurrently executing wavefronts across different CUs. These requests can then miss in the L2 TLB and travel to the IOMMU. The IOMMU thus receives a multiplexed stream of walk requests from different wavefronts.

A reasonable question to ask is how much does this interleaving potentially impact the performance? Figure 5.6 shows the potential performance cost of interleaving. The figure shows the average latencies experienced by the first- and the last-completed page walk requests from the same SIMD memory instruction. The latencies are normalized to the average latency experienced by the first completed walk request. We exclude instructions that do not generate at least two page walk
requests as they cannot interleave. Larger the latency gap, the more time an instruction potentially stalls for all of its page walk requests to complete. We observe that often the latency of the last completed walk is more than 2-3× that of the first completed page walk. This suggests that the interleaving of page walks can significantly impede forward progress.

Ideally, page walk requests should be scheduled to minimize such latency gaps. A smarter scheduler thus should strive to achieve a schedule as shown in Figure 5.4b by batching page walk requests from the same instruction. We see from the figure that load A can potentially complete much earlier without further delaying load B in Figure 5.4a.

**Key idea (2):** A smart scheduler should batch page walk requests from the same instruction to minimize interleaving due to walk requests from other instructions.

### 5.3 Design and implementation

![Diagram of page table path](image)

**Figure 5.7** Key components and actions of the SIMT-aware page table walk scheduler.

Driven by the above analyses, we propose a *SIMT-aware page table walk scheduler* in the IOMMU. Figure 5.7 shows some of the key components and actions in the IOMMU to realize such a scheduler. When an IOMMU's page table walker becomes available to accept a new request, the scheduler
selects which pending page walk request is serviced next. While we introduce a specific scheduler design, there could be several other potential designs that build upon our observations about the importance of page table walk scheduling.

Our proposed SIMT-aware scheduler follows the two key ideas mentioned in the previous section. At a high level, the scheduler first attempts to schedule a pending page walk request (in the IOMMU buffer) issued by the same SIMD instruction as the most recently scheduled page walk. If none exists, it schedules a request issued by an instruction that is expected to require the least number of memory requests (i.e., work) to service all its walk requests. For this purpose, we assign a score to each page walk request. This score estimates the number of memory requests that would be required to complete all page walk requests of the issuing instructions. The score is thus the same for all pending page walk requests generated by a given SIMD instruction. A lower value indicates fewer memory requests to service an instruction's page walk requests.

We make a few simple hardware modifications to realize the above design concept. First, each page walk request from the GPU is attached with an instruction ID (20 bits in our implementation). Correspondingly, the buffer holding the pending page walk requests at the IOMMU is extended with this ID. As shown in Figure 5.7, we then modify how the IOMMU behaves when a new page walk request arrives at the IOMMU, and when a hardware page walker becomes available to accept a new request. Below we detail actions taken during these two events.

1. **Arrival of a new page walk request:** If there is an idle hardware page walker when a new request arrives then it starts walking immediately. Otherwise, we assign an integer score (between 1 to 256, where 256 corresponds to the maximum possible number of memory accesses required if all 64 workitems need four memory accesses each to perform their respective translation) to the newly-arrived request. The score estimates the number of memory accesses needed to complete all page walk requests of the corresponding instruction. This is done in two steps. First, the new request looks up the PWCs to estimate the number of memory requests that this request alone may need to get serviced (action 1-a in Figure 5.7). This number can be between one (on a hit in all upper-levels in the PWC and thus, requiring only single memory access to the leaf-level of the page table) to four (on a complete miss in the PWC requiring the full walk of the four-level page table). Since the PWC contents could change by the time the scheduler selects the request, this number is an estimate of the actual number of memory accesses required to service the walk.

   Second, we then scan all the pending page walk requests in the IOMMU buffer to find any matching page walk requests issued by the same instruction as the newly arrived one (1-b). All requests from the same SIMD instruction have the same score. A new score is computed by adding the PWC-based score of the newly-arrived request to the previous score of an existing request in the IOMMU buffer that is issued by the same instruction. This updated score now represents the total estimated number of memory accesses required to service all the translation requests from the issuing SIMD instruction. All entries in the IOMMU that match the SIMD instruction of the
newly-arrived request (including the newly-arrived request) are then updated with this new score.

2 A hardware page walker becomes ready: When a page walk finishes, the corresponding page table walker becomes available to start servicing a new request. The scheduler decides which of the pending page walk requests (if any) it should service next. First, the scheduler scans the buffer of pending page walks to find any request that matches the instruction ID of the most recently issued page walk request (2-a). If such a request exists, it is chosen to ensure temporal batching of page walks issued by the same SIMD instruction. If no such matching request is found, then the scheduler selects a request with the lowest score. This follows the second key idea – schedule requests from the instruction that is expected to require the fewest memory accesses. Both actions are performed during the scanning of pending page walk requests (1-a). Finally, the selected page walk request is serviced as usual by first looking up the PWC for partial hits and then completing the walk of the page table (2-b).

Putting it all together: To summarize, an address translation from the GPU flows as follows. The coalescing and lookups in the GPU TLBs happen as before (refer to Section ?? for steps). The only modification for our scheduler is that each request now carries the ID of the instruction that generated it. As in the baseline, a translation request that misses in the GPU TLBs is sent to the IOMMU where it performs a lookup in the IOMMU’s TLBs. If the request misses in all the TLBs, then it is inserted into the IOMMU buffer. If any of the page table walkers (8 in the baseline) are available, then one of them starts the page walk process. Otherwise, our scheduler calculates a score for the newly-arrived request and re-scores any of the already-pending requests from the same instruction as detailed above (actions 1-a and 1-b). The request then waits in the buffer until it is selected by the scheduler.

When a page table walker finishes a walk, the scheduler selects which request it should service next. The scheduler scans pending requests in the IOMMU buffer (action 2-a) to find if there are any requests issued by the same instruction as the last-scheduled request. If so, the oldest among such requests is chosen. If not, the scheduler selects the request with the lowest score (oldest first in the case of a tie). Once a request is scheduled, the page table walker proceeds walking as usual – it looks up the PWC for partial hits before making memory accesses to the page table (2-b).

Design Subtleties: We now discuss a few intricacies of this design. First, note that the scanning of the pending page walk requests upon arrival of a new request is not in the critical path. The newly arrived request anyway queues up in the IOMMU buffer for the scheduler to select it. If a free page table walker is immediately available, the scheduler plays no role and no scanning is involved. However, it adds an extra latency in the critical path of servicing a new page walk request when the scheduler scans the pending request (2-a). Every such request in the IOMMU buffer has already suffered a long latency miss through the entire TLB hierarchy, and a walk itself requires hundreds of cycles. Therefore, the latency of scanning pending requests adds little additional delay.

As with any scheduler, the above design is susceptible to starvation. We implement an aging
scheme whereby we prioritize pending walk requests that have been passed by a large number of younger requests (in our experiments, we found that setting this threshold to two million requests worked well to avoid any potential starvation).

As briefly mentioned earlier, another subtlety is that the PWC contents may change between the time a request arrives at the IOMMU and the time the scheduler selects that request. This could lead to inaccuracies in estimating the number of memory accesses needed to service a page walk since the score is calculated when the request arrives. Unfortunately, it is infeasible for the scheduler to re-calculate scores of every pending request at the time of request selection. This would have added significant latency in the critical path. Instead, we reduce this potential inaccuracy by adding 2-bit saturating counters to the entries of the PWC. Whenever a lookup for a newly-arrived request hits in the page walk cache (1–a in Figure 5.7), the counters of the corresponding entries are incremented. The counters are decremented when a selected page walk request hits in the PWC (2–b). Thus, a value greater than zero indicates that there exists at least one pending page walk request in the IOMMU buffer that would later hit in the page walk cache when that request is scheduled. The replacement policy in the page walk cache is then modified to avoid replacing an entry with a counter value greater than zero. If all entries in a set have value greater than zero, then a conventional pseudo-LRU policy selects a victim as usual.

5.4 Evaluation

We now describe our evaluation methodology and then analyze the results in detail.

5.4.1 Methodology

We used the execution-driven gem5 simulator that models a heterogeneous system with a CPU and an integrated GPU [Gem]. We heavily extended the gem5 simulator to incorporate a detailed address translation model for a GPU including coalescers, the GPU’s TLB hierarchy, and the IOMMU. Inside the newly-added IOMMU module, we model a two-level TLB hierarchy, multiple independent page table walkers, and page walk caches to closely mirror the real hardware. We implemented different scheduling policies for page table walks, including our novel SIMT-aware page walk scheduler inside the IOMMU module.

The simulator runs unmodified applications written in OpenCL [Gro14] or in HCC [Cha16]. Table 5.1 lists the relevant parameters for the GPU, the memory system, and the address translation mechanism of the baseline system. Section 5.4.2.2 also presents sensitivity studies varying key parameters.

Table 5.2 lists the applications used in our study with descriptions of each workload and their respective memory footprints. We draw applications from various benchmark suites including Poly-
Table 5.1 The baseline system configuration.

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>2GHz, 8 CUs, 4 SIMD per CU 16 SIMD width, 64 threads per wavefront</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32KB, 16-way, 64B block</td>
</tr>
<tr>
<td>L2 Data Cache</td>
<td>4MB, 16-way, 64B block</td>
</tr>
<tr>
<td>L1 TLB</td>
<td>32 entries, Fully-associative</td>
</tr>
<tr>
<td>L2 TLB</td>
<td>512 entries, 16-way set associative</td>
</tr>
<tr>
<td>IOMMU</td>
<td>256 buffer entries, 8 page table walkers 32/256 entries for IOMMU L1/L2 TLB, FCFS scheduling of page walks</td>
</tr>
<tr>
<td>DRAM</td>
<td>DDR3-1600 (800MHz), 2 channel 16 banks per rank, 2 ranks per channel</td>
</tr>
</tbody>
</table>

bench [PY10] (MVT, ATAX, BICG, and GESUMMV), Rodinia [Che09] (NW, Back propagation, K-Means, and Hotspot), and Pannotia [Che13] (SSSP, MIS, and Color). In addition, we used a proxy-application released by the US Department of Energy (XSBench [Tra]).

In this work, we focus on emerging GPU applications with irregular memory access patterns. These applications demonstrate memory access divergence [Cha14b; Bur12] that can bottleneck a GPU’s address translation mechanism [Ves16]. However, not every application we studied demonstrates irregularity nor suffers from significant address translation overheads. We find that six workloads (XSB, MVT, ATX, NW, BCG, and GEV) demonstrate irregular memory access behavior while the remaining workloads (SSP, MIS, CLR, BCK, KMN, and HOT) have fairly regular memory accesses. Applications with regular memory accesses show little translation overhead to start with and thus, offer little scope for improvement. Our evaluation thus focuses on applications in the first category, but we include results for the regular applications to demonstrate that our proposed techniques do not harm workloads that are insensitive to translation overheads.

5.4.2 Results and Analysis

We evaluate the impact of page table walk scheduling and our SIMT-aware scheduler by asking the following questions: ① How much does the SIMT-aware page table walk scheduler speed up applications over the baseline FCFS scheduler? ② What are the sources of speedups (if any)? ③ How sensitive are the results to configuration parameters like the TLB size and the number of page table walkers?

Figure 5.8 shows the speedups of GPU applications with our SIMT-aware page walk scheduler over FCFS. The left half of the figure (dark bars) shows the speedups for irregular applications while the right half (thatched bars) shows the speedups for applications with regular memory accesses. We observe that our scheduler speeds up irregular GPU applications by up to 41%, and by 30% on average (geometric mean). On the other hand, there is little change in the performance of regular
Table 5.2 GPU benchmarks for our study.

<table>
<thead>
<tr>
<th>Benchmark (Abbrev.)</th>
<th>Description</th>
<th>Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xsbench (XSB)</td>
<td>Monte Carlo neutronics application</td>
<td>212.25MB</td>
</tr>
<tr>
<td>MVT (MVT)</td>
<td>Matrix vector product and transpose</td>
<td>128.14MB</td>
</tr>
<tr>
<td>ATAX (ATX)</td>
<td>Matrix transpose and vector multiplication</td>
<td>64.06MB</td>
</tr>
<tr>
<td>NW (NW)</td>
<td>Optimization algorithm for DNA sequence alignments</td>
<td>531.82MB</td>
</tr>
<tr>
<td>BICG (BCG)</td>
<td>Sub kernel of BiCGStab linear solver</td>
<td>128.11MB</td>
</tr>
<tr>
<td>GESUMMV (GEV)</td>
<td>Scalar, vector and matrix multiplication</td>
<td>128.06MB</td>
</tr>
</tbody>
</table>

Irregular applications

Regular applications

SSSP (SSP)          | Shortest path search algorithm                   | 104.32MB          |
MIS (MIS)           | Maximal subset search algorithm                  | 72.38MB           |
Color (CLR)         | Graph coloring algorithm                         | 26.68MB           |
Back Prop. (BCK)    | Machine learning algorithm                       | 108.03MB          |
K-Means (KMN)       | Clustering algorithm                             | 4.33MB            |
Hotspot (HOT)       | Processor thermal simulation algorithm            | 12.02MB           |

applications. This is expected; regular applications experience little address translation overhead, and thus page table walk scheduling has almost no influence on their performance. The data, however, assure that the SIMT-aware scheduling does not hurt regular workloads.

Previously in Figure 5.2 in Section 5.2, we also demonstrated how naive random scheduling can significantly hurt performance. Together, these observations show that 1) different scheduling of page walks can have severe performance implications, and 2) the SIMT-aware scheduler can significantly speed up irregular GPU applications without hurting others.

5.4.2.1 Analyzing Sources of Speedup

It is important to understand the reasons behind the observed speedups. Toward this, we first present how schedulers impact GPU stall cycles, which are the cycles during which a CU cannot execute any instructions because none are ready. Figure 5.9 shows the normalized stall cycles for each application with our SIMT-aware page table walk scheduler. The height of each bar is normalized to the stall cycles with the FCFS scheduler. A lower number indicates better forward progress since CUs are stalled for less time on average. As before, the left half shows the results for irregular applications and the right half shows those for regular applications. We observe that the SIMT-aware scheduler reduces the stall cycles by 23% on average (up to 29%) for irregular applications. This shows how the scheduler enables instructions, and consequently, corresponding wavefronts, to make better forward progress. This ultimately leads to faster execution. As expected, the stall cycles for regular applications remain mostly unchanged. Because these applications neither alter performance nor provide any new insights, the remaining evaluations in this paper focus entirely on irregular
In Figure 5.6 (Section 5.2), we showed that there can be a significant gap between the latency of the first- and the last-completed page walk for a given SIMD instruction. A larger gap indicates that instructions are waiting for a large number of translation requests to be completed, or a few requests to be completed but that are delayed due to the servicing requests from other instructions, or a combination of both effects. Our SIMT-aware scheduler batches the servicing of page table walk requests from the same instruction to reduce this gap. Figure 5.10 shows the degree of effectiveness of our scheduler in reducing the gap. Each bar represents the latency gap between the first- and the last-completed page walk requests from an instruction with our scheduler. The height of each bar is normalized to the latency gap with the baseline FCFS scheduler. As before, we exclude instructions that generate less than two page table walks as they cannot interleave. We observe that the SIMT-aware scheduler reduces the latency gap by 37% over FCFS, on average. This shows the efficacy of
Another interesting performance impact of our scheduler is that it also reduced the total number of page table walk requests. Figure 5.11 shows the number of page walk requests (i.e., number of TLB misses) with our SIMT-aware scheduler, normalized to the baseline FCFS scheduler.

We observed 21% reduction (up to 30%) in the number of page table walk requests, on average. We traced the reason for this improvement to the better exploitation of intra-wavefront locality in TLBs. Our scheduler favors SIMD instructions with lower address translation needs, which in turn aids forward progress. At the same time, our scheduler also tends to delay page walk requests from instructions that generate a large amount of address translation traffic. These high-overhead instructions are anyway likely to take a long time to complete. While the translation-heavy instructions are stalled, they are kept away from polluting (thrashing) the GPU’s TLBs. Consequently, the low-overhead instructions experience higher TLB hit rates as the useful TLB entries are not evicted.
by the high-overhead instructions. This results in a reduction in the number of TLB misses and thus, reduce the number of page walk requests.

![Normalized number of active wavefronts](image)

**Figure 5.12** Number of active wavefronts accessing the GPU's L2 TLB with SIMT-aware scheduler (normalized over FCFS).

We further validated the above conjecture by counting the number of distinct wavefronts that access the GPU’s L2 TLB over fixed-sized epochs (we used an epoch length of 1024 GPU L2 TLB accesses). Figure 5.12 presents this metric (normalized to FCFS), averaged over all epochs for the SIMT-aware scheduler. We observed a 42% reduction in the number of distinct wavefronts accessing the GPU’s L2 TLB in an epoch. This shows the role of page walk scheduling in lowering the contention in the GPU’s L2 TLB. Consequently, the number of page table walks decreases due to less potential thrashing in the TLB. This behavior has similarities to phenomena observed by others in the context of the GPU’s caches [Li15].

### 5.4.2.2 Sensitivity Analysis

We measured sensitivity of the scheduler to the GPU’s L2 TLB size, the number of concurrent page table walkers, and the size of IOMMU buffer holding the pending page walk requests.

Figure 5.13 shows the speedup achieved by our SIMT-aware scheduler with varying amounts of critical address translation resources: L2 TLB capacity and the number of page table walkers. Figure 5.13a shows the speedup with 1024 entries in L2 TLB and eight page table walkers. The average speedup achieved by the SIMT-aware scheduler over the FCFS scheduler is significant (on average, 25%) even with larger TLB. It is, however, slightly less than 30% speedup achieved with 512-entry L2 TLB. The larger TLB reduces the number of page walk requests and thus, the scope for improving performance by scheduling page walks diminishes.

On the other hand, figure 5.13b shows the speedups with 16 page table walkers. Increasing the number of page table walkers reduces the number of pending page table walks as the effective
address translation bandwidth increases. This also reduces headroom for the performance improvement achievable through better page walk scheduling. We observe that SIMT-aware page walk still speeds up applications by about 8.4% over the FCFS policy. Finally, figure 5.13c shows the combined impact of both the bigger TLB size and the increased page table walker count. In this configuration, both the increased TLB resources and the increased number of page table walkers further moderate scope for the improvement with smarter scheduling of walks. SIMT-aware scheduling speeds up applications by 5.3% in this configuration.

Overall, our SIMT-aware scheduler consistently performs better than the baseline FCFS scheduler across different configurations and different workloads, thereby demonstrating the robustness of our technique, although the amount of benefit depends on the severity of address translation bottleneck.

We then investigate the effect of IOMMU buffer size on our scheduler. The IOMMU buffer size determines the size of the lookahead for the scheduler, i.e., the maximum number of page walk requests from which it can select a request. Larger the buffer size, the larger is the lookahead potential. Figures 5.14a and 5.14b show speedups with SIMT-aware scheduler over the FCFS scheduler with 128-entry and 512-entry IOMMU buffers, respectively. All other parameters remain the same as in the baseline configuration. A smaller IOMMU buffer size reduces the opportunity for a scheduler to make smart reordering decisions, and thus, the speedups due to SIMT-aware scheduling are reduced to 13% (Figure 5.14a) with a 128-entry buffer. On the other hand, if the size of the buffer is increased to 512 entries, the average speedup jumps to 50% (Figure 5.14b). In short, the magnitude of the performance benefit from SIMT-aware scheduling varies across configurations but remains substantial across all cases.

5.5 Discussion

Why not large pages? Large pages map larger ranges of contiguous virtual addresses (e.g., 2MB) to contiguous physical addresses. They can reduce the number of TLB misses by mapping more
Figure 5.14 Speedups with varying IOMMU buffer size.

of memory with the same number of TLB entries. However, large pages are far from a panacea as decades of deployment and studies in the CPU world have demonstrated [Bas13; Gan14; Kar15]. As memory footprints of applications continue to grow, today’s large page effectively becomes tomorrow’s small page. Thus, techniques that help improve performance with small (base) pages remain useful for future workloads with larger memory footprints, even with larger page sizes. Even workloads with memory footprints of a few hundred MBs (Table 5.2) can benefit significantly from our SIMT-aware page walk scheduler, and workloads with more realistic footprints will continue to benefit from more efficient page walk scheduling, even with large pages. Unfortunately, exorbitant simulation time prevents us from evaluating such large memory footprint.

More importantly, irregular GPU applications tend to exhibit low spatial locality [Cha14b; Bur12] where large pages tend to have limited benefits. These applications see less benefit from large pages because the approach fundamentally relies on locality to enhance the reach of TLBs [Bas13]. Previous works further demonstrated that large pages can even hurt performance in some cases due to the relatively lower number of entries in large page TLBs [CB17; Bas13]. Recent works on GPUs have also demonstrated that large pages can significantly increase the overhead of demand
Interactions with Other Schedulers: In a GPU, wavefront (warp) schedulers play an important role in leveraging parallelism and impact cache behavior [Rog12; Rog13]. Previous work has also shown the importance of TLB-aware wavefront scheduling [Pic14]. Apart from the wavefront scheduler, memory controllers sport sophisticated scheduling algorithms to improve performance and fairness [Cha14b; MM08]. A reasonable question to ask is how these schedulers interact with the page walk scheduler.

Page walk schedulers play an important role in reducing address translation overheads, which none of these other schedulers aim to do. Thus, even in the presence of sophisticated wavefront and memory controller schedulers, we expect that improvements to page walk scheduling will still be useful. The page walk scheduler is unlikely to have significant interactions with the memory schedulers as the maximum amount of memory traffic from the page walk schedulers would still only consume a relatively small fraction of a GPU’s total memory bandwidth. That said, there still could be opportunities for better coordination among the different schedulers, but we leave such explorations for future work.

5.6 Related work

Three research domains are related to this work: TLB management, scheduling in memory controllers, and work scheduling in GPUs.

5.6.1 TLB Management

The emergence of shared virtual memory (SVM) between the CPU and the GPU as a key programmability feature in a heterogeneous system has made an efficient virtual-to-physical address translation for GPUs a necessity. Lowe-Power et al. [LP14] and Pichai et al. [Pic14] were among the first to explore designs GPU MMU. Lowe-Power et al. demonstrated that coalescer, shared L2 TLB and multiple independent page walkers are essential components of an efficient GPU MMU design. Their design is similar to our baseline configuration. On the other hand, Pichai et al. showed the importance of making wavefront (warp) scheduler to be TLB-aware.

More recently, Vesely et al. demonstrated on real hardware, that a GPU’s translation latencies can be much longer than that of a CPU’s and GPU applications with memory access divergence may bottleneck due to address translation overheads [Ves16]. Cong et al. proposed TLB hierarchy similar to our baseline but additionally proposed to use a CPU’s page table walkers for GPUs [Hao17]. However, accessing CPU page table walkers from a GPU could be infeasible in a real hardware due to longer latencies. Lee et al. proposed a software managed virtual memory to provide an illusion of a large memory by partitioning GPU programs to fit into the physical memory space [Lee14].
Ausavarungnirun et al. showed that address translation overhead could be even larger in the presence of multiple concurrent applications on a GPU [Aus18]. They selectively bypassed TLBs to avoid thrashing and prioritizing address translation over data access to reduce overheads. Yoon et al. demonstrated the significance of address translation overheads in the performance of GPU applications and proposed to employ virtual caches for GPUs to defer address translation only after a cache miss [Yoo18].

Different from these works, we demonstrate the importance of (re-)ordering page table walk requests and designed a SIMT-aware page table walk scheduler. Most of these works are either already part of our baseline (e.g., [LP14]) and/or are largely orthogonal to ours (e.g., [Aus18]).

Address translation overheads are well studied in CPUs. To exploit page localities among threads, Bhattacharjee et al. proposed inter-core cooperative TLB prefetchers [BM10]. Pham et al. proposed to exploit naturally occurring contiguity to extend effective reach of TLB [Pha12]. Bhattacharjee later proposed shared PWCs and efficient page table designs to increase PWCs hits [Bha13]. Cox et al. have proposed MIX TLBs that support different page sizes in a single structure [CB17]. Barr et al. proposed SpecTLB that speculatively predicts address translations to avoid the TLB miss latency. Several others proposed to leverage segments to selectively bypass TLB and the cost of TLB misses [Bas13; Gan14; Kar15]. While some of these techniques can be extended to GPUs, page table walk scheduling is orthogonal to them. Basu et al. and Karakostas et al. also proposed ways to reduce energy dissipation in a CPU’s TLB hierarchy [Bas12; Kar16].

5.6.2 Scheduling in Memory Controllers

Memory bandwidth has become a potential performance limiter with the emergence of large multi-cores and GPUs [Rog09]. Rixner et al. introduced early memory scheduling policies to exploit memory parallelism for better performance [Rix00]. Chatterjee et al. proposed staged reads that parallelize read and write requests through two staged read operations and scheduling of writes to take advantage of them [Cha12]. Yoongu et al. introduced ATLAS that prioritizes threads with least serviced at the memory controller during an epoch [Kim10a].

A GPU’s SIMT execution exacerbates memory bandwidth bottleneck [Ble11]. Ausavarungnirun et al. proposed staged memory scheduling to exploit locality by batching row buffer hit requests [Aus12]. Chatterjee et al. proposed a memory scheduler batching requests from the same wavefronts to solve memory access divergence [Cha14b]. Our SIMT-aware scheduler bears similarity with this work as we also batch requests but in the context of page walks. Further, Li et al. proposed to prioritize memory accesses with higher inter-core locality [LA16].

The fairness of resource sharing is also important in presence of multiple contenders. Mutlu et al. proposed STFM that estimates the slowdown of threads due to sharing the DRAM and prioritizes requests from the slowest thread [MM07]. PAR-BS provides QoS by batching and scheduling
requests from the same thread [MM08]. Yoongu et al. proposed TCM that groups threads with similar memory access patterns and apply different scheduling policies for different groups [Kim10b]. Jog et al. proposed to allocate fair memory bandwidth among concurrently executing kernels on different CUs in a GPU [Jog14]. Jeong et al. proposed a QoS-aware scheduling that prioritizes CPUs with low latency while guaranteeing QoS of GPUs [Jeo12]. Usui et al. proposed DASH that considers the deadline for accelerators instead of always prioritizing CPU workloads [Usu16].

These works focus solely on memory (DRAM), and not on page table walks. However, the existence of such a rich body of work shows the potential of significant follow-on research in exploring various policies for page table walk scheduling for both performance and QoS.

5.6.3 Work Scheduling in GPUs

Smart scheduling of work in the GPU’s compute units has been widely investigated, too. Rogers et al. proposed CCWS that limits the number of active wavefronts on computer units if it detects thrashing on L1 cache [Rog12]. The authors then extended it considering L1 cache usage in wavefront scheduling to reduce the impact of memory access divergence [Rog13]. Li et al. extended CCWS to also allow bypassing the L1 cache for selected wavefronts when shared resources have additional headroom after limiting wavefronts [Li15]. Kayrian et al. dynamically throttled parallelism in CUs based on application characteristics and contention in the memory subsystem [Kay13]. Unlike these works, we focus on page table walk scheduling. However, an interesting future study could explore interactions between page walk scheduling and scheduling at CUs.
As we reach the end of transistor scaling, improving the performance of modern systems encounters multiple limitations. In this trend, we have discussed interesting emerging technologies which currently attract computer architecture researcher's interests. Every technology that we have discussed introduces high potentials to relieve such limitations. However, the unique properties of new technologies also introduce new optimization challenges in modern systems. In this thesis, we discussed three emerging technologies and proposed four new optimization techniques for each new challenge.

In the first chapter, we have presented Dense footprint cache (DFC), a new last level cache design utilizing stacked DRAM. DFC minimizes the SRAM budget for the LLC tag array by utilizing large blocks (Mblocks) of a few KBs in size. It uses prediction of which 64-byte blocks within an Mblock will be used by the processor and only fetch them into the cache. These predicted-useful blocks are stored in a compact way, allowing capacity-efficient utilization of the LLC. Due to the variable actual Mblock space in the cache, DFC introduces new challenges in designing the replacement and placement policies. We have proposed three new replacement policies (LRU+, MaxAvg, and MaxMin), derived their properties analytically, and evaluated them with simulations. We have also proposed placement fit policies (best fit vs. worst fit), and early footprint update (EFU) policy. Through simulation studies with Big Data applications, we showed that DFC reduces LLC miss ratios by about 43%, speed ups applications by 9.5%, while consuming 4.3% less energy on average.

In the second chapter, we have discussed Intel's PMEM persistency instruction support for
non-volatile main memory (NVMM). We rewrote several data structures and kernels commonly found in databases and file systems, to incorporate failure safety. We showed how transactional semantics using write-ahead logging can be used as the basis for reasoning about failure safety in the context of the PMEM model. Then, we discovered performance bottlenecks resulting from the use of PMEM instructions, arising from the pipeline stalling for the completion of the *sfence-pcommit-sfence* instruction sequence. Based on these observations, we proposed speculative persistence (SP), architectural support to speculatively execute past long latency persist barriers to hide their latency and reduce their impact on performance. Speculative execution is triggered when a persist barrier stalls the pipeline. Rather than waiting, in SP, a checkpoint is taken at the persist barrier, the *sfence* is speculatively retired, and the processor proceeds speculatively retiring the *sfence* and following instructions. Meanwhile, the *pcommit* completes non-speculatively in the background. We discussed unique challenges that arise in designing SP, and architecture support that address them. We evaluate our new architecture and compared it against the same system without speculation. Our experiments show that SP achieves execution time overheads of only 4.5% on average, compared to code with logging but without PMEM instructions. This is a significant reduction compared to the 21% average overheads without speculation.

In the third chapter, we have described a new logging approach, *Proteus* for durable transactions that achieves the favorable characteristics of both prior software and hardware approaches. Like software, it has no hardware constraint limiting the number of transactions or logs available to it, and, like hardware, it has very low overhead. Our approach introduces two new instructions: *log-load* creates a log entry by loading the original data, and *log-flush* writes the log entry into the log. We add hardware support, primarily within the core, to manage the execution of these instructions and critical ordering requirements between logging operations and updates to data. We also propose a novel optimization at the memory controller that is enabled by a persistent write pending queue in the memory controller. We drop log updates that have not yet written back to NVMM by the time a transaction is considered durable. We compared our design against ATOM [Jos17] and a software only approach. Our experiments show that *Proteus* improves performance by 1.44-1.47× depending on configuration, on average, compared to a system without hardware logging and 9-11% faster than ATOM. We also show that our design performs closely to the ideal case. A significant advantage of our approach is dropping writes to the log when they are not needed. On average, ATOM makes $3.4\times$ more writes to memory than our design.

Finally, we demonstrate the importance of reordering page table walk requests for GPUs. The impact of this reordering is particularly severe for irregular GPU applications that suffer from significant address translation overheads. We observed that different SIMD memory instructions executed by a GPU application could require vastly different numbers of memory accesses (*work*) to service their page table walk requests. Our SIMT-aware page table walk scheduler prioritizes page table walks from instructions that require less work to service and further batches page walk
requests from the same SIMD instruction to reduce GPU-level stalls. These lead to 30% performance improvement for irregular GPU applications through improved forward progress. While we here proposed a specific SIMT-aware scheduler to demonstrate how better page table walk scheduling is valuable, we believe there exists scope for significant follow-on research on page walk scheduling akin to the rich body of work in memory controller scheduling.

Throughout our researches, we have noticed the possibility of future technology innovations from better optimized computer architecture designs with emerging technologies. However, I believe a lot more still remains to be investigated in these subjects, and our investigation just started. Hopefully, this thesis inspires following researchers to further explore better future systems.


