Optimizing the Use of Heterogeneous Memory

XIAOXIANG WU

Supervisor: Willy Zwaenepoel
Associate Supervisor: Baptiste Lepers, Shuaiwen Song

A thesis submitted to fulfil
the requirements of the degree of
Doctor of Philosophy

School of Computer Science
Faculty of Engineering
The University of Sydney
Australia

19 February 2026

Statement of originality

This is to certify that the content of this thesis is my own work. This thesis has not been submitted for any other degree or purpose.

I certify that the intellectual content of this thesis is the product of my own work, and that all assistance received in preparing this thesis and all sources have been acknowledged.

Xiaoxiang Wu

Acknowledgements

I would like to thank my advisors Willy Zwaenepoel, Baptiste Lepers, and Shuaiwen Song for their continuous guidance and support throughout my PhD candidature. I am also grateful to my team member Yuben Yang for all his help, and to Jane Cusack and Lyndon McKevitt, my advisors' colleagues, for their much-appreciated administrative assistance.

My thanks go to the members of the FSA Lab, Haojun Xia, Donglin Zhang, Zhongzhu Zhou, and Fengxiang Bie, for their active participation in discussions and for exchanging ideas with me. I also wish to thank the members of the Inria KRAKOS Lab in Grenoble, especially Alain Tchana, Annie Simon, Maxime Collette, Jordan, and Kenta Ishiguro, as well as members of the Inria Lab in Paris, notably Jean-Pierre Lozi and Julia Lawall. These meetings were a great pleasure and deeply inspiring for my research in systems.

I appreciate the funding support from the Australian government and PRSS, and particularly thank those involved in these programs, including Alan Fekete.

Thank you to the anonymous reviewers and to all those who will read this thesis.

Finally, thanks my parents, Xiaoli Pan and Xihong Wu, my girlfriend, Zhijie Huang, and my cats, Bobo Wu and Johnny Sheldon. I could not have come this far without them.

Author Attribution Statement

Chapter 2 'Things you wanted to know about persistent key-value stores but were afraid to ask' of this thesis has two publication attempts, but it is not yet published. I designed and conducted the study, analysed the data and wrote the drafts of the manuscript.

Chapter 3 'Pre-Stores: Proactive Software-guided Movement of Data Down the Memory Hierarchy' of this thesis has been published in EuroSys 2025 as [155]. I designed and conducted the study, analysed the data and wrote the drafts of the manuscript.

Chapter 4 'Opportunities With Emerging High Speed Unified Memory' contains ongoing original research that has not yet been published. I designed and conducted the study, analysed the data and wrote the drafts of the manuscript.

In addition to the authorship attribution statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Xiaoxiang Wu

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Willy Zwaenepoel

Artificial Intelligence

No content produced by generative AI tools has been used in the preparation of this thesis.

Xiaoxiang Wu

Australian Government Support

This research was supported by an Australian Government Research Training Program (RTP) Scholarship, Australian Research Council Grant DP21010198.

Abstract

My thesis investigates trade-offs and novel techniques in the design and optimization of data-intensive systems across persistent memory (PMem), heterogeneous memory hierarchies, and unified physical memory architectures.

I will start by reviewing and challenging the existing understanding of persistent memory, where we investigate past approaches and check their validity against present hardware. Building on top of that knowledge, we introduce a technique called Pre-stores to optimize the use of various heterogeneous memory in a generic way. We propose novel techniques to proactively move data, improving performance by up to 2.3×. Moving beyond that, we explore next-generation high-speed unified memory. We conduct thorough performance investigations, implement key-value stores specifically optimized for this architecture, and share insightful findings along the way.

To be more exact, first, in chapter 2, we study persistent key-value stores and isolate the impact of individual design techniques within a unified code base. Unlike prior works that evaluate complete systems, our methodology enables an apples-to-apples comparison of trade-offs. We show that random allocation achieves performance comparable to log-structured persistence while avoiding garbage-collection latency spikes, that persistent CPU caches, such as eADR (Extended Asynchronous DRAM Refresh) or CXL (Compute Express Link) global flush, often hinder rather than help performance, necessitating explicit flushes, and that recovery mechanisms require careful handling of allocator metadata, with transactions imposing nontrivial overhead.

Second, in chapter 3, we introduce the concept of software pre-storing, the converse of prefetching, which issues instructions to proactively move data down the memory hierarchy. Implemented via existing CPU instructions, pre-storing benefits write-intensive workloads, especially on architectures with heterogeneous memories such as PMem or CXL-attached

DRAM. We develop DirtBuster, a tool that identifies applications and code regions where pre-storing is beneficial. Evaluations on ARM and x86 systems with PMem and cache-coherent DRAM demonstrate performance improvements of up to 2.3× across key-value stores, HPC applications, message-passing systems, and TensorFlow.

Finally, in chapter 4, we examine unified memory architectures that combine high-bandwidth access with a coherent, shared address space, thereby addressing the limitations of conventional iGPU (bandwidth-bound) and dGPU (PCIe-bound) designs. Using a state-of-the-art unified memory architecture platform, we characterize performance under diverse workloads, identify scenarios where unified memory architectures excels, and reveal the costs of fully shared memory. Our analysis provides practical guidelines for memory management in unified memory architectures systems and highlights their significant potential for balanced CPU–GPU workloads.

Contents

Statement of originality ii

Acknowledgements iii

Author Attribution Statement iv

Artificial Intelligence v

Australian Government Support vi

Abstract vii

Contents ix

List of Figures xii

Chapter 1 Introduction and Overall Discussion 1

Chapter 2 Things you wanted to know about persistent key-value stores but were afraid to ask 4

2.1 Introduction 4

2.2 Experiment setup 7

2.3 Favoring sequential accesses over random accesses 7

2.3.1 Performance of updates on a database with a small number of items 9

2.3.2 Performance of deletes on a database with a small number of items 12

2.3.3 Additional notes on performance 13

2.3.4 Implications 14

2.4 Recovery tradeoffs 15

2.5 Evaluation 19

2.6 Generalization of the results 23

2.6.1 Future of Optane PMem 24
2.7 Related Work 25
2.8 Conclusion 27

Chapter 3 Pre-Stores: Proactive Software-guided Movement of Data Down the Memory Hierarchy 28

3.1 Introduction 29
3.2 Pre-stores 31
3.3 The diversity of memory architectures 32
3.4 Example uses of pre-stores 34
3.4.1 Problem #1 - The random order of evictions (on Machine-A) 34
3.4.2 Problem #2 - Delayed cache operations (on Machine-B) 38
3.5 Potential pitfalls of pre-stores 41
3.6 DirtBuster: design and implementation of a tool for pre-stores 42
3.6.1 Overview 43
3.6.2 Implementation 45
3.6.2.1 Step 1: Detecting write-intensive functions 45
3.6.2.2 Step 2: Analyzing memory access patterns 46
3.6.2.3 Step 3: Re-read and re-write distance 47
3.7 Evaluation 48
3.7.1 Setup 48
3.7.2 Improving sequentiality (on Machine A) 49
3.7.2.1 Machine learning workload 49
3.7.2.2 HPC workloads 53
3.7.2.3 Key-value stores 55
3.7.3 Improving latency (on Machine B) 58
3.7.3.1 Key-value stores 58
3.7.3.2 Message passing workload 60
3.7.4 Overhead of pre-stores 61
3.7.4.1 Pre-stores suggested by DirtBuster 61
3.7.4.2 Incorrect manual use of pre-stores 62

3.7.4.3 Manual intervention and edge cases 62
3.8 Related Work 63
3.9 Conclusion 65

Chapter 4 Opportunities With Emerging High Speed Unified Memory 67
4.1 Introduction 67
4.2 Background 71
4.2.1 Discrete GPU 71
4.2.2 Unified Memory 73
4.2.3 Memory Speed 76
4.3 Hash tables 81
4.3.1 CPU-only 83
4.3.2 GPU-only 86
4.3.3 Combined CPU and GPU 87
4.3.3.1 Approach 1: Handing threads to the GPU 88
4.3.3.2 Approach 2: Using GPU idle time 89
4.3.3.3 Issues with approach 1 and 2 91
4.3.3.4 Approach 3: Replicating the hash table 91
4.3.4 Prefetching and its effects 95
4.3.5 Concurrently updating hashtable 97
4.4 Analytics Query 100
4.5 Related Work 103
4.6 Conclusion 104

Chapter 5 Final Remarks and Future Work 105

Bibliography 106

List of Figures

2.1 Update throughput, varying the value size and the corresponding write throughput. 9

2.2 Per thread throughput when using Logs. The GC threads preempt the main KV threads, causing fluctuations in throughput. 12

2.3 Delete throughput, varying the value size and the corresponding write throughput. 13

2.4 Update and delete throughput with a large index. Logs still handle bursts of requests faster than allocators, but the difference is less pronounced than on a small database. 14

2.5 Left: Performance relative to minimal persistence (%). Burst update throughput, depending on the persistence guarantee. Right: Recovery time breakdown. 18

2.6 YCSB sustainable throughput. Using our allocator, which persists data at using random writes, outperforms logs which use sequential writes. Persisting the index up to is 50% slower than keeping it in DRAM. Using transactions dramatically impacts performance in update intensive workloads. 19

2.7 Minimal persistence (allocator) performance.Top: YCSB A Bottom: YCSB C. Values either kept in the CPU caches or flushed. 21

3.1 Topology of Machine B. 34

3.2 Content of a 2-way cache after writing four arrays, one after the other. Each array is presented in a different color. Because the cache evicts data in 'random' order, a given array may be partially evicted multiple times. 35

3.3 Machine A. (a) Improvement brought about by activating the *cleaning* pre-store call in Listing 2, varying the element size. (b) Write amplification with and without *cleaning*. 37

3.4 Demoting data forces the CPU to write the data to its caches. Without the demote instruction, the CPU can keep the modified `a` private, until it is forced to commit the change, causing delays in the execution of the CPU pipeline. 37

3.5 Machine B executing Listing 3. Relative performance improvement brought by demoting the dirty data before the fence. Demotion improves performance by up to 65%. The higher the latency of the cached device, the larger the window of time during which demotion improves performance. 40
3.6 DirtBuster relies both on sampling and binary instrumentation. Sampling allows us to find the write-intensive functions, and binary instrumentation allows us to analyze the access patterns of these functions (sequential writes, writes before a fence, and re-read and re-write distances). 43
3.7 Machine A. TensorFlow. Performance improvement brought by pre-storing data. As advised by DirtBuster, cleaning the cache improves performance, while skipping the cache decreases performance. 53
3.8 Machine A. TensorFlow. Adding a cleaning pre-store significantly reduces write amplification. 53
3.9 Machine A. Normalized runtime of the NAS benchmarks. Lower is better. 55
3.10 Machine A. CLHT performance, in requests per second. Higher is better. 57
3.11 Machine A. Masstree performance, in requests per second. Higher is better. 57
3.12 Machine A. Write amplification of CLHT executing YCSB A (lower is better). 58
3.13 Performance of CLHT on Machine B-fast (FPGA configured with a low latency) and Machine-B-slow (high-latency FPGA). 59
3.14 Performance of Masstree on Machine-B-fast and -slow. 59
4.1 Illustration of a common machine with discrete GPU 71
4.2 Illustration of Apple's Mac Studio M2 Ultra 74
4.3 Time taken for GPU to sort an integer array. Lower the better.
Left: Discrete GPU Right: Integrated GPU. Tests beyond 12 hundred million keys are omitted for discrete GPU as VRAM is already exhausted. 75
4.4 Sequential read(top) write(bottom) speed, higher the better. 77
4.5 Latency for pointer chasing, lower the better. 79
4.6 Speed of copying data, higher the better. Using library memcpy(), each of 1GB data. 80
4.7 Design of Mega-KV, a GPU only hash table 82

4.8 CPU hashtable performance, ported from Mega-KV. Using 16 threads, hashtable with 100 million keys. Uniform means random distribution of keys, where zipfian means skewed distribution of keys. 84
4.9 GPU hashtable performance, ported from Mega-KV. Varying batch size, hashtable with 100 million keys. Note: Performance of Mega-KV is copied from respective paper directly, due to unsuccessful launch using newer machines. 86
4.10 Compare of time to launch GPU kernel between M2 Ultra and CUDA. Lower the better. 87
4.11 Design of extended hash table 87
4.12 Hashtable throughput, using a total of 16 worker threads, varying number of GPU worker threads. When it increases, the number of CPU worker threads automatically goes down. Batch size for GPU worker threads is always large (100000 keys). 'Expecting' is thus the sum of two. 88
4.13 The actual bench marked throughput, derived from figure 4.12. The observed throughput is consistently lower than expected throughput. 89
4.14 Per thread time spent on running a batch of 100000 lookup. Derived from curve of GPU from figure 4.12 89
4.15 Hash table throughput comparison between approach 1 and 2. Higher the better. 90
4.16 Hash table throughput of approach 1 and 2 when using separated memory region. Higher the better. 91
4.17 Design illustration of a duplicated hash table to improve performance with unified memory. When running lookups, each device accesses its own hash table. On the other hand, updates are replicated across both hash tables to ensure consistency. 92
4.18 A smarter hashtable design. Instead of duplicating, CPU holds a smaller hash table with keys that frequently being accessed for optimal latency. GPU holds a larger hash table and complete big batches for high throughput. Memory footprint is therefore much lower. 93
4.19 Benchmark test of changing the size of hot data which CPU will be working on. The size of data for GPU is unchanged. 94

4.20 Hash table throughput with respect to number of prefetched buckets. Higher the better. 95
4.21 Performance of approach 2 with Prefetch added, 20 buckets are prefetched in advance. 96
4.22 The design of running an analytics query on M2 Ultra. 101
4.23 Time breakdown of running listing 14. Lower the better. Configurations are same as table 4.4. 102

CHAPTER 1

Introduction and Overall Discussion

With changing computational workloads and the increasing sophistication of today's applications set, there is a growing need for heterogeneous memory architectures. In general, developers have had to settle for a combination of volatile DRAM storage, which is fast and has low latency, as well as slower, block-based storage such as SSDs or HDDs. DRAM provides performance but with data volatility. Storage devices provide persistence, but have much lower bandwidth, high access latency and coarse-grained accesses compared with DRAM, and are not byte addressable.

This fundamental disconnect between speed and resilience has resulted in a large research effort for synchronizing data across tiers, taking point-in-time snapshots, and data placement for performance or fault tolerance. But progress until now hasn't enabled that ascension to be realized, and the hierarchical divide has persisted between the two memory tiers—until now, that is, with byte-addressable, non-volatile persistent memory (PMem). These new modules boast performance similar to DRAM while delivering the endurance of storage. PMem enables simpler and more scalable design options especially for persistence aware data structures and systems which were not practical with complex copying or synchronization.

But even with PMem, the capacity of a single memory device is still inadequate to feed the appetite of an enterprise-scale application. Growing DRAM is difficult and expensive, so there are some systems that are designed as a hybrid with more than one memory technology to increase capacity. Popular configurations now include not only DRAM and PMem, but also SSDs, remote storage, pooled memory accessed over RDMA or DMA paths, and even memory borrowed from a remote system. Keeping track and coordinating access over such diverse, geographically dispersed memory can be very complicated.

This is where memory abstraction layers like Compute Express Link (CXL) can be utilized. With a consistent, device-agnostic programming interface, CXL allows to access a uniform memory space populated by diverse memory interconnects in a cache-coherent manner. It hides low-level details like the diversity of latencies among systems. This simplifies that programming and helps fully utilize hardware. However, abstraction layers pose a series of challenges: by abstracting away physical implementation-specific details, they might obscure a potentially large execution cost in case of performance irregularities, and constrain the efficient access strategies developers may like to exploit for high-efficiency operations.

Another strong pressure on memory architecture comes from the emergence of large language models(LLMs) and other AI workloads. Such models tend to demand huge VRAM memory sizes in order to perform at scale, but the size of the GPU memory is still severely limited, even while suited-for-model size system memory (DRAM or PMem) is orders of magnitude larger. Conventional offloads utilizing memory of the CPU side lead to unacceptable overhead because of the slow data transfer or non-coherent access destination. In response, there's a trend toward High-Bandwidth Unified Memory Architectures, where different compute units (GPUs, NPUs, etc.) can access large pools of memory at near-native speed, bypassing the overhead and potential duplication of both data and latency.

These evolving design points not only expose and increase memory capacity and flexibility, but they also open up new paths for system design—especially in applications like keyvalue stores, databases, and other data-intensive systems that require lowlatency, high-throughput memory access across compute boundaries.

However, the emerging memory design also throws the new challenge. Efficient utilization of heterogeneous memory systems is still a challenging task. In this thesis, we examine these suboptimal memory traffic patterns in environments with heterogeneous memory. We pinpoint scenarios where access is not fully optimal (due to misalignment, cache inefficiency, bandwidth underutilization) and suggest potentially widely-applicable techniques for obtaining the most optimized access in heterogeneous memory landscapes.

The following are the specific contributions of this thesis:

- Chapter 2 is a complete paper that contrasts different designs of PM-aware key-value stores to assess how well they fare against new PMem features.
- Chapter 3 is a paper accepted at EuroSys 2025, that brings techniques that can improve the write pattern and the write throughput and latency.
- Chapter 4 is a work-in-progress paper looking at the next generation of unified high bandwidth memory.

Taken together, these work lay the groundwork for memory-aware systems that can effectively take advantage of the potential of newer memory hierarchies – performance, capacity, and coherence in an increasingly heterogeneous world.

CHAPTER 2

Things you wanted to know about persistent key-value stores but were afraid to ask

---

Discussion: In the past, numerous approaches were proposed to accommodate slow memory, many of which are still in use today. However, with the improvement of memory speed, some of these techniques are no longer effective. This fact is rarely recognized, largely due to the lack of fair comparisons conducted on modern hardware. As the first chapter of my thesis, we revisit these approaches and provide an apple-to-apple comparison by integrating them into a single framework.

2.1 Introduction

Key-value stores (KVs) are the storage engines for many Internet services. Many systems have been developed and are in production use [94, 122, 41, 50], and many papers have been written on their performance [8, 98, 4, 14, 40, 23]. Recently, the storage eco-system has been disrupted by the arrival of fast storage with high bandwidth, high IOPS and low latency – NVMe SSDs [91], persistent memory [74, 17, 66, 80], and CXL storage extensions [29]. Researchers have proposed new key-value store designs to take advantage of this new hardware developments [7, 88, 12, 11, 25, 27, 68, 72, 78, 89, 91, 102, 113, 114, 117, 124].

These papers typically present a new system and compare its performance (and occasionally its persistence guarantees) to previous systems. While interesting, these comparisons are often difficult to interpret, because it is not clear which aspects of the new system contributed to its alleged better performance. Variations in hardware and software infrastructure further obscure the comparison. In this chapter, we take a different approach. Rather than comparing

different systems, we compare the effect in isolation of different techniques used in building key-value stores.

In this chapter, we set out to evaluate the effect of the following key design choices:

- Many published KVs persist their data in logs to favor sequential accesses. We study the trade-offs of using random or sequential allocation of data.
- An increasing number of devices allow their memory to be mapped in an application's address space and cached by the CPU (GPUs, PMem, CXL storage, ...). We explore the impact of having persistent CPU caches (e.g., eADR domain for PMem, Global Persistent Flush for CXL [126]). Persistent caches allow simplifying the design of KVs by freeing the developers from the need to manually flush data to persistent storage, but they drastically change the lifetime of data in the CPU caches. We explore the performance trade-offs of having persistent data kept in the caches.
- Finally, existing KVs vary in the amount of data that they persist. On one side, some KVs only persist their keys and values, and have to rebuild their index and associated metadata after a crash. On the other side, some KVs try to persist enough data to allow instant recovery after a crash. We study the trade-offs of the various approaches on performance and recovery time.

To this end, we have implemented these techniques in a single code base. We then selectively enable the different techniques, and compare the performance of the resulting approach on the same hardware platform.

Since the only commercially available implementation of persistent caches is for PMem, we evaluate the designs on a machine equipped with PMem. However, most conclusions presented in this chapter only depend on the fact that PMem allows persisting data with a high IOPS and low latency (we discuss this point in Section 7).

To precisely pinpoint the effect of these decisions, we first present measurement results that exercise each of the common API methods of key-value stores, updates, inserts, deletes and

lookups. We explain the overall results using detailed lower-level measurements. In addition, we also provide overall results using the YCSB benchmarks.

Some of our results are surprising. These results constitute the main contributions of this chapter:

- We show that, contrary to the assumptions made in previous works [27, 93, 13, 145, 97, 109, 127, 68, 163, 158, 65], the extra sequentiality provided by logs has little impact on the overall performance of a KV. We show that logs perform better in bursty update- and delete-intensive workloads because they erase values from storage asynchronously. We show that in the long run, the garbage collection operation used to compact logs negatively impact performance and persisting values using random allocations actually performs better than using logs.
- Surprisingly, except when working with small value sizes, it is faster to flush caches than to keep dirty persistent data in the caches. We show that unflushed dirty persistent data can significantly slow down *DRAM* accesses, and negatively impact the performance of KVs.
- The choice between keeping the index in DRAM or having it persistent involves a rather obvious tradeoff between failure-free performance and recovery time, but the magnitude of the failure-free performance hit to achieve instant recovery is substantial. We also show that most existing KVs do not persist enough data to allow instant recovery. Most notably, most KVs do not persist the metadata of their persistent allocator, which can lead to substantial memory leaks if no recovery procedure is implemented.

The outline of the rest of this chapter is as follows. In Section 2.2, we describe our experiment setup. In Sections 2.3 to 5, we present the main tradeoffs explored in this chapter: in Section 2.3 between random and sequential allocation, and in Section 2.4 the recovery trade-offs. In Section 2.5, we present results for the YCSB benchmarks. We discuss the generalization of our results on systems other than PMem in Section 2.6, we cover related work in Section 2.7 and conclude in Section 2.8.

2.2 Experiment setup

Hardware configuration. Unless stated otherwise, all the experiments presented in this chapter are run on a two-node NUMA machine, with 56 Intel Xeon Gold 6230 cores running at 2.0GHz (28 cores per node, hyperthreading disabled), 256GB of DRAM, and 4*128GB Intel Optane NV-DIMMs on socket 0.

Software configuration. We explore different key-value stores designs, but all the explored designs execute the same codebase. The main differences are the amount of persisted data, whether data is persisted using a standard allocator or using logs, and flushed or kept in caches.

In this work, we explore the impact of these choices on performance. Our goal is to provide high-level observations that are not specific to exact software implementation details, but rather fundamental trade-offs implied by these choices. That being said, we have profiled and optimized the index, allocator and logs to ensure that they match the best performance reported in the related work [19, 27, 85, 145, 139].

Keys are stored in a concurrent B+Tree index, which we either keep in DRAM or in persistent storage. The index associates each key with the location of its value in storage. In storage, each value is persisted with a timestamp (to distinguish between versions of an item in case of a crash) and a checksum (to allow detecting partially persisted items in case of a crash). When the index is not persistent, we also persist the key to allow rebuilding the index.

2.3 Favoring sequential accesses over random accesses

Designs. Current KVs broadly fall into two categories. Some KVs persist their data in logs to favor sequential accesses to storage, while other KVs persist data at random locations to avoid the cost of garbage collection inherent to sequential allocations. In this section, we explore the trade-offs of these two designs.

Implementation. When using logs, data is written to storage sequentially. Historically, KVs used large logs to reach the maximal sequential speed of hard disks (e.g., LSM-trees typically store data in logs larger than 100MB)[52]. The more recent developments of log-based KVs favor smaller logs, to limit the amount of data that needs to be scanned and compacted during garbage collection[27, 93, 13, 145]. Our implementation follows these recent developments: each KV thread owns a 4MB log, and new values are appended at the end of the log. When a log is full, the KV thread requests a new log from the system. We keep track of the percentage of each log's space that is occupied by outdated items. When a value is updated, the new value is appended at the end of the current log, and we increase the percentage of outdated space of the old value's log. Deleting a value consists in appending a tombstone in the current log, and increasing the percentage of outdated space of the old log. Once half of a log is occupied by outdated items, we add the log to the garbage collection queue. The garbage collector reads the logs of its queues and checks which items are current and which are outdated. Checking if an item is current is done by comparing its location in storage with the location stored in the index (the index always points to the most recent version of an item). The GC appends all current items to a new log, and updates the index to refer to their new location. The logs in the GC queue are then marked as free, and will later be reused by the main KV threads. When using logs, all accesses to the storage are done sequentially.

When persisting data at random locations, KVs rely on persistent memory allocators to allocate memory. Our allocator manages persistent memory using superblocks like most state-of-the-art (persistent) allocators [19, 37, 16, 105]. A superblock is a 4MB memory region separated into slots of a given size, which are used to store items of a given size range (e.g., items of size 40-64B are allocated in a superblock partitioned into 64B slots). Each superblock contains metadata and a list of available slots. When allocating an item of size X, we return the first available slot in the superblock that contains items of size X. If no available slot is found, a new superblock is allocated. When freeing an item, the item is marked as freed in storage (by clearing its checksum) and then its location is inserted in the list of available slots of its superblock. After deleting random items or updating items (with consists of inserting a new value and deleting the old value), new allocations mostly reuse previously freed slots, and happen at 'random' locations.

In this section, the medatada (free lists of superblocks, percentage of outdated space in logs, etc.) is maintained in DRAM. We explore the impact of persisting the metadata in Section 5.

2.3.1 Performance of updates on a database with a small number of items

We first evaluate the performance of allocation vs. logs on a database with a small number of items (10 million keys), varying the value size. We place the index in DRAM. Using a small index in DRAM allows us to minimize the impact of the index, to stress the impact of the value storage on performance.

Figure 2.1 presents the throughput of updates. For logs, we present two throughputs: the peak throughput that the KV can handle during short bursts and the 'sustainable throughput': the throughput computed by waiting for the garbage collector threads to finish their job (we configured the number of GC threads to the best value we could find for each experiment).

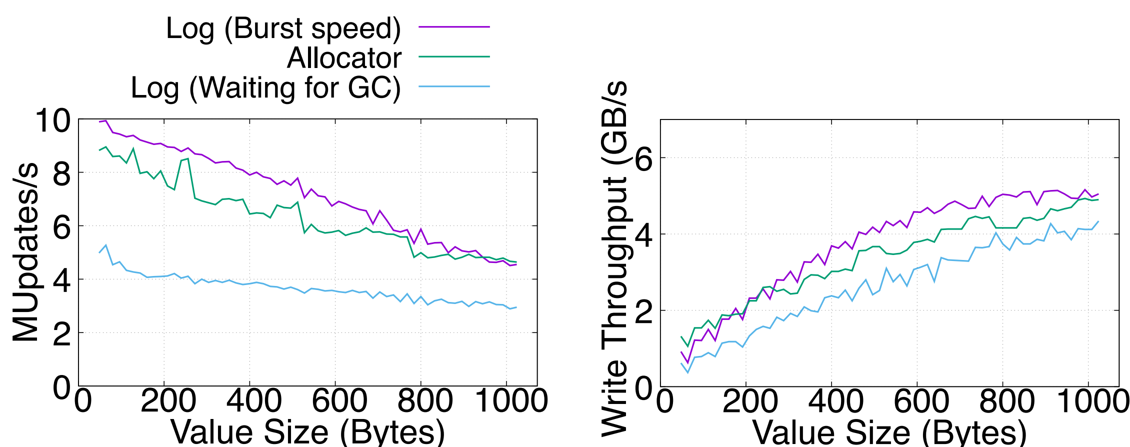

FIGURE 2.1: Update throughput, varying the value size and the corresponding write throughput.

The peak throughput of logs is up to 25% faster than the throughput persisting data at random locations, and logs are overall faster for values up to 1000B. The throughput of all designs decreases as the value size increases because more data is persisted. The performance of logs

decreases smoothly as the value size increases, and the performance of our allocator evolves in 'steps' because allocated items are stored in slots hosting a given size range.

Why are logs faster in bursty workloads? Table 2.1 presents the time spent in the three main operations performed when doing an update: updating the index, finding a spot for the new value (and freeing the old value for the allocator), and writing the new value. Table 2.1 does not include the garbage collection time for logs because the garbage collection is not triggered on short bursty workloads.

Surprisingly, with small values, the performance difference mostly comes from the difference in the time spent in the allocator's vs. log's management functions (41% of the time for the allocator vs. 32% with logs). The exact percentages presented in Table 2.1 are implementation dependent, but updating a value is fundamentally less costly using logs than using an allocator. In a log, the update is complete once the new value has been appended to a log. In the allocator, the old value needs to marked as free (= clearing a checksum in our case, which causes a write to storage) prior to the new value being written. The superior burst performance of Logs is thus explained by the asynchronous deletion of old values. Counter-intuitively, the sequentiality provided by logs has little impact on performance because writing the value only represents a small percentage of the total time.

With large values, both the Logs and the allocator spend most of their time writing values. The time spent in the log's functions increases slightly with value size because of the more frequent recycling of logs (the bigger the value, the more frequently logs need to be recycled because a larger percentage of their content becomes outdated). The extra time spent marking values as free in the allocator vs. the time spent recycling logs becomes equivalent, and both solutions perform equally. Unsurprisingly, both the allocator and the logs spend the same amount of time writing values to storage – writing 1 KB values can be done at the same bandwidth as writing data sequentially and both designs reach the maximum throughput of the device (see Figure 2.1).

Why are logs slower in sustained workloads? Under sustained load, a KV cannot indefinitely postpone the deletion of old values. We measure the 'sustainable throughput' as the throughput

(a)

Time in ns (percent)	64B	128B
Index Update	1064 (59.57%)	1064 (57.73%)
Allocator's functions	399 (22.34%)	418 (22.68%)
Write Value	323 (18.09%)	361 (19.59%)

(b)

Time in ns (percent)	64B	128B
Index Update	1064 (66.67%)	1064 (62.92%)
Log's functions	304 (19.05%)	323 (19.10%)
Write Value	228 (14.29%)	304 (17.98%)

(c)

Time in ns (percent)	256B	512B	1024B
Index Update	1064 (55.45%)	1064 (44.80%)	1064 (33.71%)
Allocator's functions	437 (22.77%)	437 (18.40%)	446 (14.15%)
Write Value	418 (21.78%)	874 (36.80%)	1645 (52.14%)

(d)

Time in ns (percent)	256B	512B	1024B
Index Update	1064 (58.64%)	1064 (45.16%)	1064 (33.73%)
Log's functions	361 (19.90%)	437 (18.55%)	456 (14.46%)
Write Value	389 (21.47%)	855 (36.29%)	1634 (51.81%)

TABLE 2.1: Breakdown of the time spent when doing an update, when values are (a/c) stored using a memory allocator or (b/d) stored using logs. In parentheses, the percentage of the total time spent updating the index vs. in the allocator's/log's function vs. writing the value. With small values, most of the time is spent traversing and updating the index and in the allocator's/log's functions.

obtained by bounding the size of the GC queues. The performance of Logs, waiting for the GC, is up to $2.2\times$ slower than that of using an allocator.

Performing garbage collection is much slower than freeing data because the garbage collector needs to figure out, for each item that it is garbage collecting, if the item is current or outdated. Checking if an item is current requires a lookup in the index, a costly operation [145]. Profiling reveals that the GC threads are bound by the latency of pointer chasing operations performed by lookups in the index (even though, in this experiment, the index is stored in DRAM).

Performance over time. Garbage collection threads interfere with the main KV threads. Figure 2.1 presented the average throughput of the basic KV operations, but it should also

be noted that GC threads may cause significant fluctuations in the performance over time. Figure 2.2 presents the evolution of the throughput over time, measured on a core, when updating items. After 0.5s of execution, the throughput drops and then fluctuates. The fluctuations are explained by the garbage collection running on the same cores as the KV threads, and pre-empting them. We measured the maximum pre-emption period of the KV threads to be 26ms. Garbage collection can thus be an issue for workloads with strict SLAs.

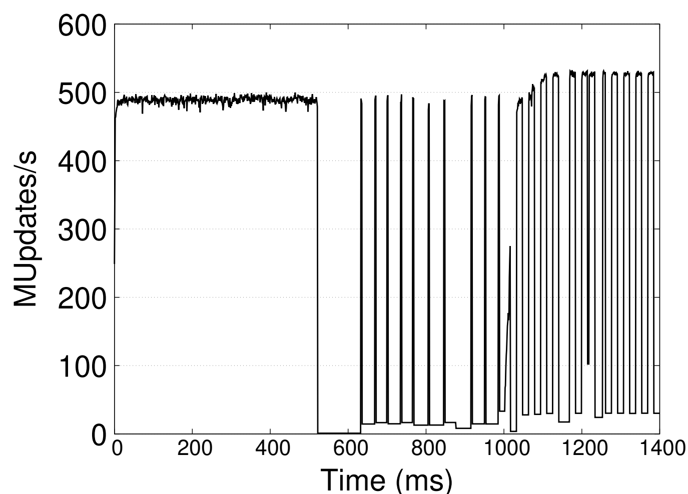

FIGURE 2.2: Per thread throughput when using Logs. The GC threads preempt the main KV threads, causing fluctuations in throughput.

2.3.2 Performance of deletes on a database with a small number of items

Figure 2.3 presents the throughput for delete operations. Logs outperform the allocator when performing short bursts of delete operations. The main reason is that deleting a value using a log simply consists of appending a tombstone in a log and updating the index, while the allocator frees the item and then deletes the key from the index. In general, deleting a key from the index is more costly than just updating a key (because the deletion operation may cause merge operations of the nodes of the index). However, just as is the case for updates, when waiting for the garbage collector to actually delete the old values, Logs are up to $4.5\times$ slower than allocators.

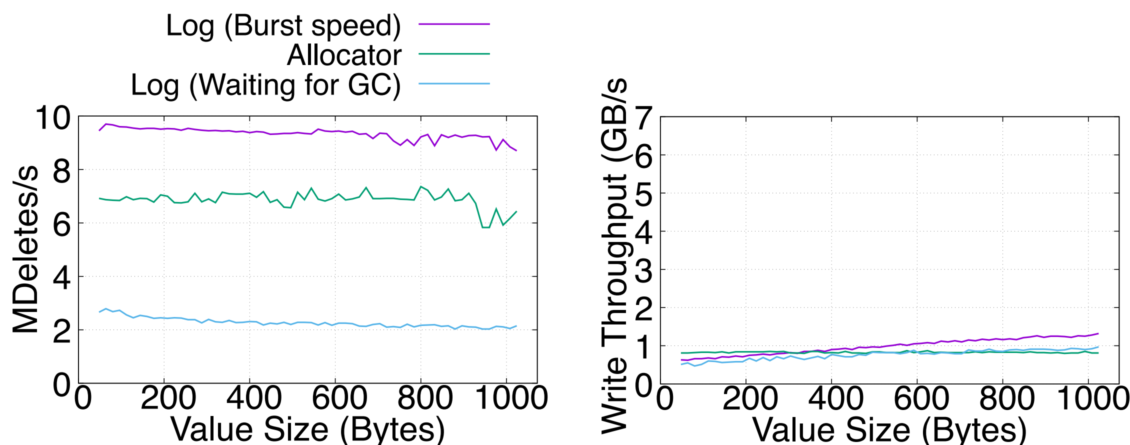

FIGURE 2.3: Delete throughput, varying the value size and the corresponding write throughput.

2.3.3 Additional notes on performance

In the previous experiment, we compared the performance of Logs and allocators using a small index, to stress the influence of value storage on performance. The influence of storage decreases with index size. Figure 2.4 presents the performance of Logs and allocators when allocating 100M items. The advantage of Logs to handle small bursts decreases as the size of the database increases: logs handle bursts of update requests up to 19% faster (vs. up to 25% with a small index) and delete requests up to 33% faster (vs. up to 42% with a small index). The allocator is $2-4\times$ faster than Logs when taking GC into consideration.

We focused our analysis on the performance of updates and deletes. Insert operations have the same performance when values are persisted using Logs vs. using an allocator. The performance of Logs is equal to its bursty speed (no deletion happens, so the garbage collector threads do not run), and the allocator operates at its maximum speed (no freeing of value happens, and persisting items using random writes is not limiting performance). Lookups perform the same when values are persisted using Logs vs. using an allocator.

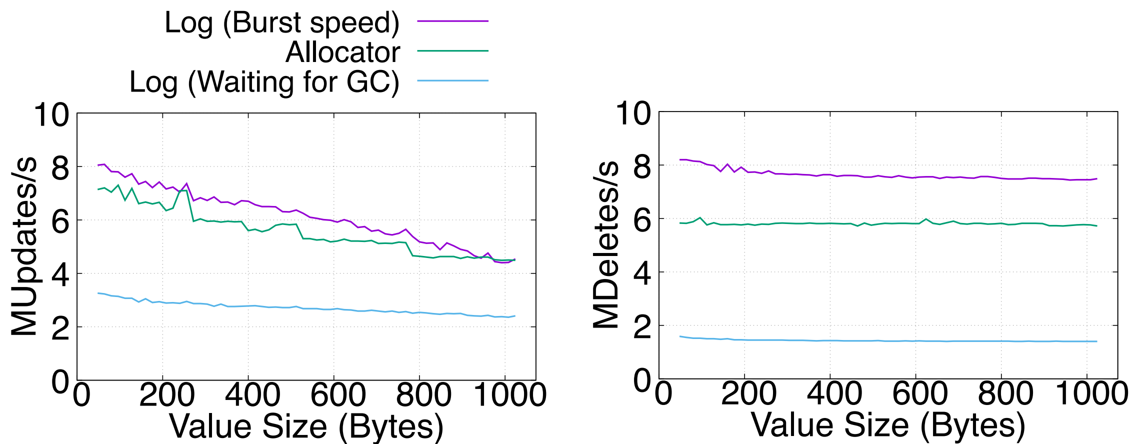

FIGURE 2.4: Update and delete throughput with a large index. Logs still handle bursts of requests faster than allocators, but the difference is less pronounced than on a small database.

2.3.4 Implications

Logs are the default choice on storage with low IOPS. Surprisingly, even on fast storage, logs remain advantageous in bursty update and delete intensive workloads. Counter-intuitively, the performance gain mostly comes from the asynchronous deletion of outdated values, and not from the sequentiality provided by logs. We believe this observation to be novel, and to go against the assumptions made by previous work. Indeed, most published work justify logs using sequentiality arguments [27, 93, 13, 145, 97, 109, 127, 68, 163, 158, 65], but we found that, for key-value stores, writing data sequentially only provides minor benefits in most workloads. When updating small values, most of the time is spent in the index and allocator (so the extra sequentiality has little performance impact). When updating large values, allocators can saturate the device bandwidth, so the sequentiality of logs provide no extra performance.

We also showed that, while beneficial in the short run, asynchronous deletions can create interferences and latency spikes, and reduce the overall sustainable throughput.

2.4 Recovery tradeoffs

Designs. Previous works vary in the amount of data they persist. Some KVs only persist their keys and values, and rebuild their index after a crash [27, 117, 156, 180, 162, 145]. Most previously published persistent key-value stores persists their keys, values and index [88, 68, 89, 102, 113, 179, 147, 7, 139, 85]. Some KVs also persist their persistent allocator's metadata [72, 37, 133, 118, 42, 85]. In this section, we study the impact of these three designs on performance and recovery time.

Listing 1 presents the typical workflow of inserting a value in a key-value store. The value is first allocated, modified, flushed, and finally a pointer to the value is inserted in the store. It is possible for the system to crash while performing the allocation, or just after allocating the value, but before inserting it in the tree. Such crashes may result in inconsistent allocator metadata, or persistent memory leaks (unreachable persistent memory region that cannot be used after a crash). To circumvent these issues, most persistent memory allocators either offer a transactional interface [72, 37, 133, 118, 42, 85] (using shadow copy or bitmasks), or recovery procedures to assess the reachability of persisted data [19] (i.e., a post-crash garbage collection phase).

Listing 1 Operations performed while inserting a key

```
void *value = alloc_on_storage(value_size);
value = ...;
flush_to_persist(value);
tree->insert(key, value);
```

Fully measuring recovery time is thus more complex than just keeping the index in persistent memory. We evaluate the three main designs decisions that lead to recovery trade-offs:

- *Full persistence* uses transactions to keep a consistent state in storage. Index, values and the allocators' metadata are persisted.
- *Index persistence (with allocator or logs)* only persists the index, keys and values, but requires a scan of the data to recover the allocator's or logs' metadata.
- *Minimal persistence (with allocator or logs)* only persists the keys and value and requires a scan of the data to rebuild the index and the allocator's or logs' metadata.

Implementation. Persistent allocators rely on 'root pointers' to recover their state[19, 37, 16, 105]. Root pointers are named pointers that can be recovered after a crash. From these pointers, a developer can retrieve reachable pointers and inform the allocator of their reachability. From this set of reachable pointers, the allocator infers the set of unreachable items in its slabs, and marks them as free. The un-reachability computation is because the reachability list has to be fully computed first before being able to infer the un-reachability list.

Recovery using logs is simpler: the only missing information that needs to be recomputed is the percentage of outdated data in each log. This can be done by scanning the logs and by checking if the scanned value is outdated, current, or replacing a value scanned in a previous log. The recovery still requires scanning all the items in the logs and comparing their location against the location present in the index.

Transactional interfaces allow wrapping the allocation and the insertion in the tree in a single atomic block. Existing implementations [71] perform transactions using shadow copy: data is first written in a log, the log is persisted, and the original values are updated. In case of a crash, the log is replayed to reach a consistent state. Recovery using transactions is near instantaneous.

We briefly describe the implementation of the various recovery procedures used by the different designs:

- Minimal persistence (with allocator): we iterate through all allocated superblocks and check which items are valid (i.e., not freed). This is done by looking at the checksum of the items (freed items have a null checksum). All non-freed items are inserted in the DRAM index and marked as reachable.
- Minimal persistence (with logs): we scan all allocated logs. For every item in a log, we check if the item is already in the index. If the item is not in the index, we insert it. If the item is already in the index, we compare the versions of the log's item vs. the item that is already in the index, and keep the most recent one. The logs metadata is rebuilt as the data is scanned.

- Index persistence (with allocator): we start from the persisted root of the tree, and explore the tree in a DFS manner. All reachable nodes and values are marked as reachable.
- Index persistence (with logs): We start from the persisted root of the tree, and explore the tree in a DFS manner. We recompute the percentage of outdated data contained in each log.
- Full persistence: the recovery is handled automatically. The recovery consists in replaying any interrupted transaction log.

Performance. Keeping the index in PMem negatively impacts performance. Figure 2.5 presents the relative performance difference between storing the index in DRAM vs. in PMem. On average, storing the index in persistent memory is 50% slower. The exact impact of storing the index in PMem may vary depending on implementation details of the index, but we believe that this observation applies broadly – regardless of the implementation, persisting the index is slower than keeping it in DRAM because of the higher latency of pointer chasing operations in storage compared to DRAM.

Full persistence (using transactions to persist index, values and allocator metadata) is $9\times$ slower than minimal persistence. Profiling shows that the performance difference is mostly explained by the extra overhead of having a transaction-safe allocator. Indeed, supporting transactions at the allocator level requires serializing some operations (e.g., a freed pointer can only be reused only once the transaction that performed the free operation completes). Profiling reveals that, when using Full persistence, the KV spends 70% of its time in locks used to serialize access to the allocator's metadata. Our current implementation of transactions has the same performance as libpmdk[71]. It is possible that more optimized transaction managers would manage to reduce the locking overhead but, regardless of the exact implementation, using transactions always adds some overhead to an application.

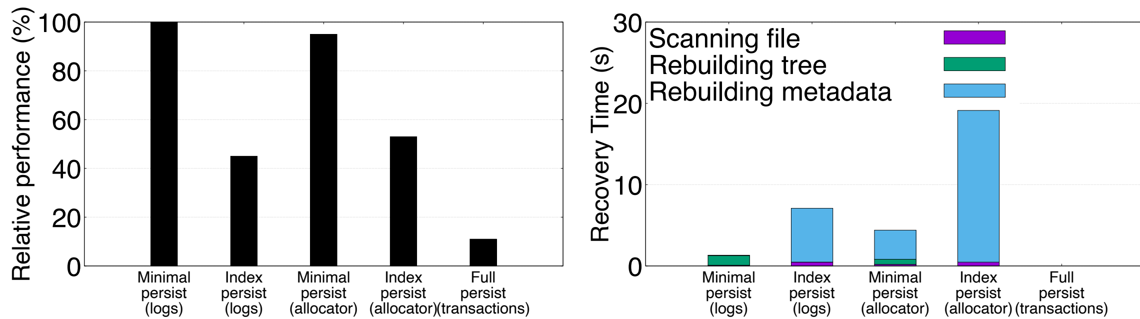

FIGURE 2.5: Left: Performance relative to minimal persistence (%). Burst update throughput, depending on the persistence guarantee. Right: Recovery time breakdown.

Recovery time. The cost of recovery varies greatly between solutions. Figure 2.5 presents the breakdown of recovery time.

When the allocator's metadata is volatile, most of the recovery time is spent rebuilding the allocator's metadata. More specifically, 95% of the recovery time is spent rebuilding reachability lists. Logs spend the majority of the recovery time accessing the index to check which values are current and which values are outdated. Regardless of the persistency guarantee, logs allow faster recovery because less metadata needs to be rebuilt. Unsurprisingly, using transactions allows the KV to recover almost instantly (few cycles in most runs).

A surprising observation is that it is *faster* to recover when *not persisting* the index than when persisting the index. This counter-intuitive observation is explained as follows:

- When values are persisted using logs, rebuilding the logs metadata requires querying the index to infer which values are current and which values are outdated. When the index is in PMem, this operation is costly. It is faster to rebuild the log metadata plus index, when the index is in DRAM (minimal persist) rather than just rebuilding the logs metadata when the index is in PMem (index persist).
- When values are persisted using an allocator, rebuilding the allocators metadata requires building the list of reachable items in every superblock. When the index is in PMem, more data is persisted in PMem, so more reachability lists need to be computed, explaining the higher total recovery time [19].

Additional notes on performance. In this section, we have discussed the recovery trade-offs in case of a crash. However, the proposed designs may also be used to achieve persistence in case of orderly shutdowns of the application. For orderly shutdowns, persisting the persistent allocator's metadata is also required for instant recovery, but doing so does not require the use of transactions – a consistent allocator's state can be persisted just before shutdown. In that situation, keeping the index persistent allows for a faster shutdown time than keeping the index in DRAM, and has a lower impact on performance than using transactions.

Implications All the designs but transactions require scanning the data to recover in case of a crash. Most surprisingly, just persisting the index without persisting the allocators metadata comes with a big performance hit, and provides no benefit in terms of recovery time.

2.5 Evaluation

In this section, we evaluate the various designs using a YCSB workload. We run the workload with 256B values, and 10M or 100M keys. Figure 2.6 presents the performance results. The results confirm the observations made in the previous sections.

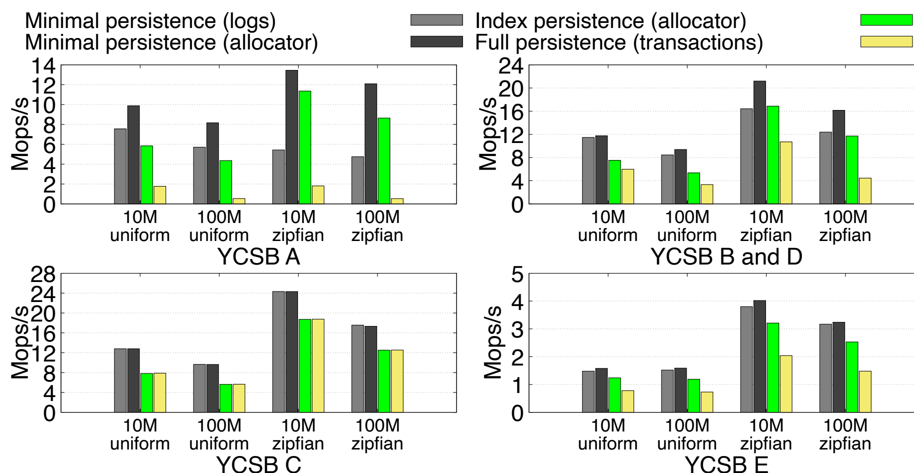

FIGURE 2.6: YCSB sustainable throughput. Using our allocator, which persists data at using random writes, outperforms logs which use sequential writes. Persisting the index up to is 50% slower than keeping it in DRAM. Using transactions dramatically impacts performance in update intensive workloads.

YCSB A. YCSB A performs 50% updates and 50% reads. Unsurprisingly, having the index in DRAM and values persisted using an allocator outperforms other designs. Logs suffer from GC overheads. The performance difference between allocator and logs is more pronounced when running the zipfian YCSB workload. Indeed, in the zipfian workload the same keys are frequently updated, resulting in faster index lookups (most of the accessed nodes of the index reside in the CPU caches) – data is thus updated more frequently, resulting in more pressure on the garbage collection threads.

Similarly to the observations of Section 5, the performance is halved when the index is persisted. The lower performance is explained by the higher latency of the pointer chasing operations required to perform lookups and updates in the index. The performance is divided by up to 22 when using transactions (with 100M keys and a zipfian distribution). The lower performance is explained by the lock overhead of transactions. In conclusion, aiming for lower recovery time has a huge toll on performance.

YCSB B and D. YCSB B and D perform 5% updates and 95% reads. The allocator and Logs have similar performance on the uniform workloads, but the GC overhead is noticeable in the zipfian workload. Even when updating values only 5% of the time, when the updates happen at a high rate, the GC threads still negatively interfere with the main KV threads.

Persisting the index still significantly impacts performance because of the higher latency of pointer chasing operations. Keeping the index in DRAM is up to $1.5\times$ faster in the uniform workload, and $1.8\times$ in the zipfian workload. Transactions impact performance less in YCSB B than in YCSB A because read queries do not run in transactions (so most queries have no transaction overhead), but still induce significant overheads (transactions are up to $3.6\times$ slower than minimal persistence).

YCSB C. YCSB C performs 100% reads. In this workload, allocator and Logs have similar performances because lookups perform the same operations whether data is persisted using the allocator or Logs. Persisting the index still negatively impacts performance because of the extra latency of pointer chasing operations ($1.6\times$ slower on uniform and $1.3\times$ slower on Zipfian compared to not persisting the index).

YCSB E. YCSB E performs 95% scans and 5% updates. The conclusions on YCSB E are similar to that of YCSB B, but the impact of persisting the index is slighly lower (scanning the index stresses PMem less than performing random lookups).

Influence of persistent caches. Figure 2.7 presents the impact of keeping data in caches on the performance of YCSB A (update intensive) and YCSB C (read only).

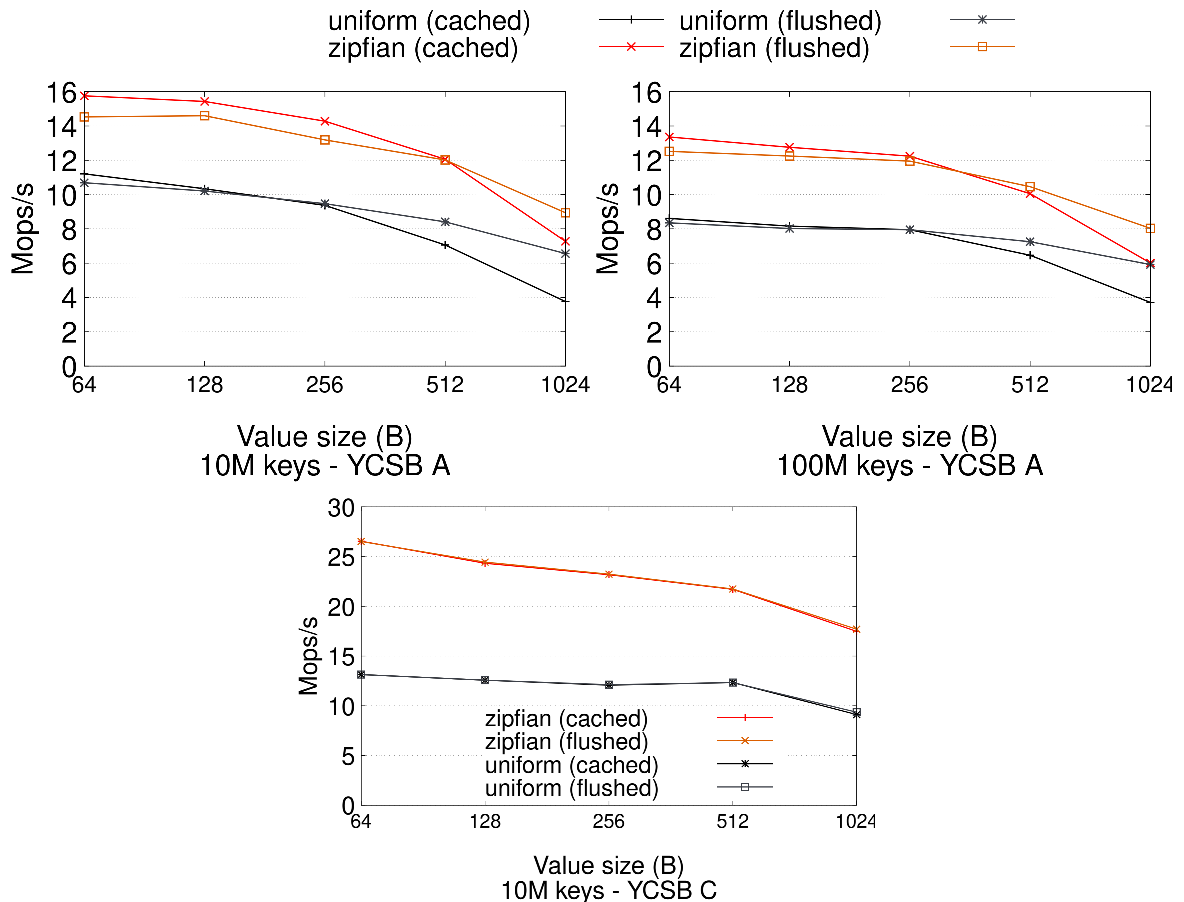

FIGURE 2.7: Minimal persistence (allocator) performance.Top: YCSB A Bottom: YCSB C. Values either kept in the CPU caches or flushed.

In YCSB A, when keys are accessed uniformly, flushing dirty values from the caches matches or outperforms keeping dirty data in the caches. When working with small values (64B), profiling shows that the writeback queues of the processor are rarely full, and the number of stalled cycles is the same when flushing the caches manually or when keeping data in

the cache (evictions do not block processing because they can be enqueued in the writeback queue). When working with larger values, the processor tries to evict data faster than it can be processed by the writeback queues, causing the CPU to stall. Similarly to the experiment of Section 4, updated values are rarely re-read, and keeping them in the cache causes the CPU to stall on performance critical paths of the KV.

When keys are accessed in a Zipfian manner, keeping the updated values in the cache performs slightly better than flushing them for small value sizes, but not for large values. Profiling shows that the performance is a combination of three factors:

- When the keys are accessed in a Zipfian manner, doing lookups in the index only cause few cache misses and, as a consequence, few cache evictions ($3\times$ less than in the uniform workload). Lookups in the index are thus less impacted by dirty data in the cache when running a zipfian workload than an uniform workload.
- The smaller the value size, the lower the number of index nodes that are evicted between lookups.
- The smaller the value size, the more values can be re-read from the cache when re-updating a value or reading a previously updated value.

These three factors result in the writeback queues of the processor having free slots most of the time when working with values less than 512B and a small index (left graph of Figure 2.7). The larger the index, the more its performance suffers from the presence of dirty data in the caches. When executed with 100M keys, YCSB only benefit from persistent caches with values less than 256B (right graph of Figure 2.7).

We note that, regardless of the configuration, persistent caches rarely bring significant performance gains (at most 14% performance improvement), but can dramatically impact performance in some configurations (flushing is $1.5\times$ faster when working withh 1024B values and a uniform distribution). Unless working with known and predictably skewed workloads, persistent caches thus provide little performance benefits.

Unsurprisingly, YCSB C is not impacted by persistent caches because YCSB C does not create any dirty persistent data in the cache. Evictions from the cache do not require a writeback and do not impact the lookups in the index.

Conclusion. The observations made in the previous sections also apply to the YCSB workload. Logs suffer from the overhead of garbage collection, keeping dirty data in the caches is detrimental to performance when not re-used, and persisting the index has a large impact on performance.

2.6 Generalization of the results

In this chapter, we studied the performance and correctness trade-offs of KVs on persistent memory. We believe that the observations of this chapter apply more broadly.

Sequential accesses vs. random accesses. KVs are currently more limited by the time they spend in the index and allocator's/log's functions than by the limited speed of random storage access. This conclusion is a consequence of the high IOPS and low latency of PMem, but is not PMem-specific. As the speed of storage continues to increase, IOPS and bandwidth are unlikely to limit the performance of KVs.

Keeping data in caches. Delaying the flushing of dirty data from caches can negatively impact performance because it causes the CPU to stall. We believe that this conclusion broadly applies to all systems that are latency sensitive and delay the persistence of data. For instance, previous work [91] has shown that the software buffers, traditionally used in disk-based KVs to delay the persistence of dirty data, negatively impact performance. It is interesting to see that the same conclusion applies to the CPU caches when working at high request rates.

To avoid the overhead of kernel I/O syscalls, it is likely that future storage devices will allow mapping their memory in the applications' address spaces (it is already the case for some GPUs, PMem and accounted for in the CXL specification). In this context, it is likely that the writeback of dirty data from the CPU caches to the persistent medium will negatively impact

performance, especially if the evictions happen in critical paths of the applications. Carefully managing the timing of evictions is thus likely to still matter in future KVs.

Recovery cost. Regardless of the storage medium, it is worth considering the performance/recovery tradeoffs of a KV. This chapter presents the counter-intuitive result that persisting the KV index, without persisting the KV allocator's metadata, is counter-productive for performance and for recovery. For a KV to allow instant recovery, all inconsistencies have to be detectable and solved without scanning the data. Most notably, rebuilding an allocator's metadata requires scanning all the persisted data items, which has an inherent storage and CPU cost, and should be avoided for a KV to claim 'a negligible' recovery time.

2.6.1 Future of Optane PMem

Even though Optane PMem is no longer commercially available, the findings in this chapter are not unique to this hardware. We make our observations with its persistence, higher bandwidth and lower latency. Table 2.2 lists upcoming devices such as ones attached to CXL, they may have even better performance while maintaining persistence.

TABLE 2.2: Different future CXL devices that offer similar performance to PMem

Technology	Persistent	Granularity	Latency	Bandwidth
PMem	Yes	Cacheline	300 ns	6̃–15 GB/s per module
CMM-H CXL SSD [167]	Yes (NAND-backed)	Cacheline/block	0.6 µs (mem) / 10 µs (block)	2 GB/s (mem) / 4 GB/s (block)
CXL-SSD Type-3 [87]	Yes	Byte addressable via CXL.mem	µs-scale	Few GB/s (PCIe-limited)
Persistent CXL switch [60]	Yes (remote persistent pool)	Cache line	DRAM-class + network hop	Tens of GB/s fabric aggregate

2.7 Related Work

Indexes. Numerous hash [66, 102, 113, 114, 179, 180] and range indexes [7, 25, 68, 89, 117, 147, 162] designed have been proposed in prior work. Many range indexes [72, 68, 89, 102, 113, 179, 147, 7] view PMem as an extension of DRAM, and persist the index. Other designs, such as FP-tree [117], HiKV [156], SOFT [180], FlatStore [27] and NV-Tree [162] keep the routing nodes in DRAM and only persist data, trading better latency for increased recovery time. Bullet [66] takes a different approach, by maintaining two copies of the index: a copy is kept in DRAM to provide good latency, and a copy is kept in storage to provide good recovery time. The persistent copy is updated off the critical path. A number of existing techniques do not sort the information in the leaves, sacrificing sequential range scans access for lower maintenance [117, 162]. Most of these KVs require a scan of the data to rebuild the allocator's metadata. Strongly consistent allocators [37, 133, 118, 42, 85] persist bitmaps to allow faster recovery time while sacrificing performance.

Recent work also proposes general techniques to adapt existing work to PMem. RECIPE [88] provides a framework to convert DRAM indexes to PMem, ensuring correctness, scalability, and crash consistency. While RECIPE is straightforward to apply, it cannot be used by all DRAM indexes. For instance, RECIPE assumes that mechanisms for memory reclamation are in place, which is not guaranteed by all concurrent DRAM data-structures, and it does not support wait-free techniques (e.g., read retries). FLEX [159] is a library that replaces conventional file operations with similar DAX-based operations. FLEX can be used in file systems, KVs, and database engines. PACTree took the size of PMem XPLine into consideration and designed KV specifically for PMem. NBTree [169] leverages eADR to increase performance.

Programming primitives. PMwCAS [147] is a multi-word compare-and-swap operation for PMem, used to adapt indexes such as concurrent B-trees [7, 94] and skip lists [122] from DRAM to PMem. Twizzler [17] is a new operating system (OS) that argues for persistent pointers at the OS-level, to facilitate programming with PMem.

Disk KV designs. Much recent work aims to adapt Log-Structured Merge (LSM) KVs to PMem. NoveLSM [80] tiers the LSM components between DRAM, PMem, and SSD. It reduces latency fluctuations by allowing direct updates to PMem when the LSM DRAM component is overwhelmed. MatrixKV [166] keeps the first level of the disk component in PMem and proposes a new column-wise compaction scheme to decrease write stalls. SLM-DB [78] combines B+-tree and LSM techniques. SLM-DM keeps data unsorted, stored separately in a PMem buffer. SLM-DB stores the index in PMem as well, which can lead to detrimental NUMA effects. SLM-DB performs garbage collection using LSM-inspired compaction. While it is useful to build upon established systems (e.g., LevelDB, RocksDB), this approach has the disadvantage of inheriting techniques that are inefficient in PMem, such as CPU-intensive compactions. Prism[139] uses multiple layers of storage devices, including DRAM, PMem, and SSDs to achieve high throughput while ensuring crash consistency. MioDB [48] uses byte-addressable persistent skip lists to reduce the latency of persisting data to speed up LSMs.

Reducing the impact of background threads. Previous work has been done on reducing performance fluctuations in key-value stores that use background threads. PebblesDB [124] uses fragmented LSM trees to reduce the impact of compactions. PebblesDB postpones compactions to the last level of the LSM tree. SILK proposes an I/O scheduler for LSM KVs to schedule compactions when the KV is mostly idle [11]. TRIAD [12] reduces write amplification and thus the amount of data written per compaction. ChameleonDB [172] shards the LSM tree, to split data and perform compactions on smaller datasets, reducing the durations of stalls. Under high load, all four systems eventually need to run compactions, causing latency spikes and throughput drops for client operations. Pacman[145] uses shortcuts to minimize the index traversal time and speed up garbage collection. ROART[105] uses background garbage collection to allow instant new allocations without recovery.

Random vs. sequential IOs The observation that sequential IOs do not always outperform random IOs has been made in previous work on block devices. HashKV [23] and KVell [91] persist items directly at their final location on fast NVMe drives. KVell further showed that keeping data unordered on disk is beneficial for performance, even when performing scans.

Research on NVMe caches [157] also found that it is not necessary to write data sequentially to get good performance.

2.8 Conclusion

We have evaluated the impact of three design decisions on the performance of KVs. First, we have seen that persisting data sequentially provides little benefit over persisting data using random writes, but can introduce large overheads due to the need for garbage collection to happen. Second, we have seen that keeping dirty data in the CPU cache negatively impacts performance. Third, we have explored the non trivial trade-offs between performance and recovery time, and have shown that persisting only a subset of the KVs metadata is counter-productive both in terms of performance and recovery time.

CHAPTER 3

Pre-Stores: Proactive Software-guided Movement of Data Down the Memory Hierarchy

---

Discussion: The experiments in figure 2.7 on persistent caches have revealed the necessity of explicit flushing. However, we believe that this can be done in a smarter and more generic way to benefit more applications.

Using flushes in key-value stores is straightforward because they write large chunks of data as values. However, many applications perform similar operations in different ways. For example, a simple matrix multiplication writes the results to the final matrix sequentially. If the matrices are large enough, this is effectively a replay of sequential writes similar to those in key-value stores. With that in mind, we began exploring how to flush data in less obvious applications. The initial exploration was particularly difficult. Without knowledge of the application, we can only manually read application code and blindly insert flushes, brute-forcing locations where flushes are beneficial. However, doing this inspired and motivated us towards a full solution to the problem.

At present, the demands of software applications generally outpace the advancement of hardware. Consequently, industries often adopt heterogeneous memory systems. However, employing such systems through traditional local memory abstractions can result in suboptimal performance. I regard this as the most important chapter of my thesis, as it examines current approaches to memory usage and proposes a novel, generic method to improve performance.

3.1 Introduction

In newer memory architectures, CPU caches store data from memories with a growing range of access characteristics, such as differences in bandwidth, latency or size of the transfer unit between cache and memory. For instance, recent Intel CPUs can cache data from standard DDR memory, but also from high bandwidth memory (HBM) [35], persistent memory (PMem) [32] and CXL-attached storage [33]. Furthermore, an increasing number of NUMA servers comprise standard ARM or x86 nodes interconnected in a cache-coherent manner with an accelerator or an FPGA [28].

Caches are designed with conventional DRAM in mind. We demonstrate that they perform suboptimally when they cache data for memories with characteristics substantially different from DRAM. Specifically, differences between the sizes of the cache lines and and the memory unit cause write amplification. Furthermore, high latencies can cause memory coherence operations to be delayed.

We show that cache performance can be improved by asynchronously moving data down the memory hierarchy 'earlier' than would be necessary because of memory models or resource constraints. We formalize this idea with the concept of software *pre-storing* (simply referred to as pre-storing in the rest of the chapter). A pre-store is the converse of a prefetch. While a prefetch directs the CPU to move data up the memory hierarchy [103, 86, 111, 152, 121, 110, 73, 115], a pre-store directs the CPU to move data down the memory hierarchy. We describe different forms of pre-storing, including moving data from private CPU storage to globally visible cache locations and moving data from the cache to memory.

We present application scenarios in which pre-stores provide performance benefits. First, consider a scenario in which one or more writes are followed sometime later by a fence. The CPU is allowed to keep the data written in its registers or in a private buffer, outside of the cache, until the occurrence of the fence. When hitting the fence, the CPU sends the writes to the cache, the cache reads the corresponding cache lines from memory if necessary, and updates them. These cache line reads and updates must be completed before the fence

can complete. Current cache implementations have been optimized to hide the latency of conventional DRAM, but fail to hide the latency of less conventional memories. Pre-stores direct the CPU to asynchronously initiate the writes to the CPU caches in advance, with the goal that they complete before the fence is reached.

Second, performance issues also arise when the CPU evicts data from its caches. Consider a scenario in which an application writes contiguous data items, e.g., successive elements in an array. Data is written to the cache sequentially and eventually written back to memory, but because of unpredictable cache conflicts or non-LRU cache replacement, writebacks to memory may not be sequential. The lack of sequentiality in evictions is typically not a problem for DRAM, but can negatively impact memories with large write granularities (e.g., Optane Persistent memory or CXL-based storage that internally write data in chunks of 256B or more). Pre-stores improve sequentiality of writes from the cache to memory.

We describe DirtBuster, a tool that identifies possible performance anomalies of the type discussed above, suggests locations where a pre-store can be inserted, and what type of pre-store would be most beneficial. DirtBuster relies on dynamic analysis, using a combination of memory access sampling and binary instrumentation.

We present measurement results of four sets of applications: a subset of the Phoronix benchmarks [108], including TensorFlow [107], a subset of the NAS benchmarks [10], two key-value stores [39, 106] and a message passing benchmark. We demonstrate the benefits of pre-storing on different CPU architectures and cacheable memories. Pre-storing improves performance by up to a factor of $2.3\times$.

The contributions of this chapter are:

- The identification of performance issues with the interaction between caches and (unconventional) memory systems.
- The introduction of the concept of software pre-storing.
- A tool for discovering code segments in which software pre-storing can address the aforementioned performance issues.
- An evaluation of pre-stores to mitigate the observed performance problems.

The outline of the rest of this chapter is as follows. Section 3.2 introduces the concept of pre-storing. Section 3.3 introduces the diversity of architectures and cacheable memories. Section 3.4 provides use cases for pre-stores. Section 3.5 discusses the possible overheads of using pre-stores. Section 3.6 presents the design and implementation of DirtBuster. Section 3.7 evaluates the impact of pre-stores. Section 3.8 presents the related work, and Section 2.8 concludes.

3.2 Pre-stores

We provide a function that allows moving data down the memory hierarchy:

```
prestore(void *location, size_t size, op_t op)
```

The `op` parameter is used to indicate the desired operation. It can take the following values:

(1) `demote`. A `demote` pre-store moves data down in the cache hierarchy, for instance, from the L1 cache to the L2 cache, or from private CPU buffers to the L1 cache

(2) `clean`. A `clean` pre-store directs the CPU to write dirty data from the cache to memory.

The pre-store operation acts on `size` bytes, starting from `location`. Regardless of the `op` flag, pre-stores keep the data in the cache: *cleaning* the data propagates the modifications to memory but does not invalidate the cache. The `prestore` function is non-blocking: data is moved down the cache hierarchy or written to memory in the background without blocking the CPU pipeline.

It is also possible to *skip* the cache entirely, writing a value directly from registers to memory with so-called non-temporal stores. Unlike *demotion* and *cleaning*, which can be achieved by simply inserting a suitable call to `prestore`, non-temporal stores require modifying the code in a more complicated manner.

Implementation of pre-stores

Common architectures such as x86 and ARM offer instructions that allow easy implementation of pre-stores.

Intel CPUs allow developers to move data down the cache hierarchy with the `cldemote` instruction (*demotion* operation). Data can also be moved from caches to memory using the `clwb` instruction (cache line writeback, or *cleaning*).

ARM CPUs offer a wide range of cache-related instructions. For instance, `dc cvau` directs the CPU to write data to the 'point of unification', the point at which all cores in the system see the same value (the L2 cache for most modern devices) [138].

Regardless of the architectures, all the instructions presented in this section are non-blocking, allowing for a non-blocking implementation of pre-store. In previous works, these instructions have typically been used to force the CPU to write data in a specified order, for instance, to provide a consistent representation of data to all cores, or to persist data in order. In our work, we use these instructions to direct the CPU to execute writes in the background.

3.3 The diversity of memory architectures

The classical memory architecture consists of a number of levels of cache and a single type of memory (DRAM). Given the single type of memory sitting beneath the caches, the cache design is closely integrated with the memory system.

Recently, architectures have come on the market that deviate from this classical design. Multiple types of memory can be cached by a single system of caches. In this chapter, we focus on two characteristics that we believe will become commonplace in future servers.

Differences between the CPU cache line size and the internal write granularity of the memory These differences are likely to become the norm, as caches are asked to cache more and more diverse devices. Table 3.1 presents the internal granularity of reads and writes for different CPUs and memory devices. For instance, storage uses cache lines size $4-8\times$ larger than that of Intel CPUs.

TABLE 3.1: Devices internally read and write at different granularities (e.g., 64B cache line for an Intel CPU vs. 256B for an Optane persistent memory).

Device	Internal granularity
Intel CPU	64B
ThunderX ARM CPU	128B
Optane PMem	256B
CXL SSD	256B/512B (current technologies [135])

As a currently available example of this type of architecture, we use a two-node NUMA machine, with 40 Intel Xeon Gold 6230 cores running at 2.10GHz, 128GB of DRAM, and 8*128GB Intel Optane NV-DIMMs. In this machine, the CPU caches data at a 64B granularity, but the Optane memory internally writes data at a 256B granularity. We refer to this configuration as *Machine A*.

Weak memory architectures and long latency memories

With the current trend towards architectures with disaggregated memory, caches are required to front memories with longer latencies than directly attached DRAM. Byte-addressable memory accessible over CXL [129] is a prime example of such architectures, but other examples include accelerators for AI, streaming and networking workloads [69, 34, 1] and cache-coherent FPGAs [28].

As a currently available instance of this type of architecture, we use an Enzian [28] prototype, referred to as *Machine B*. Enzian is an asymmetric NUMA system with a 48-core ArmV8 ThunderX-1 and a Xilinx XCVU9P Ultrascale+ FPGA. The FPGA's memory is transparently cached by the CPU. Figure 3.1 depicts the topology of machine B. The applications run on the CPU, which transparently caches the FPGA's memory.

The latency and bandwidth of the FPGA can be configured. To illustrate the diversity of devices that will be accessed in a cache-coherent manner, we test two configurations of machine B: a lower-latency configuration, in which the FPGA's memory is accessed in 60 cycles at 10GB/s (representative of future high-end CXL-accessible memory), and a higher-latency configuration in which the FPGA is accessed in 200 cycles at 1.5GB/s (representative

of medium-tier CXL-accessible storage). We refer to these two configurations as *Machine B-Fast* and *Machine B-Slow*, respectively.

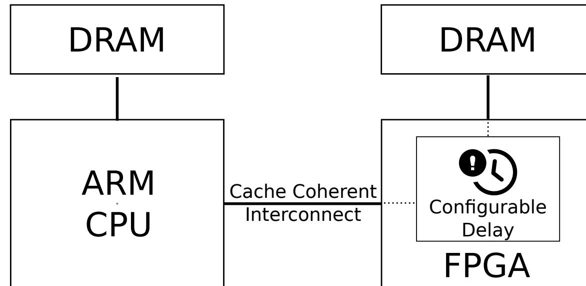

FIGURE 3.1: Topology of Machine B.

3.4 Example uses of pre-stores

This section provides simple examples of the use of pre-stores and measurements to illustrate their benefits.

3.4.1 Problem #1 - The random order of evictions (on Machine-A)

Modern CPU caches tend to evict data in a seemingly random order. Even when data is written sequentially by the application, the corresponding cache lines may not be written out in the same order to memory. We explain why this behavior impacts performance.

Context CPU caches are associative: the cache is divided into bins, and each bin can hold n cache lines (n-way associative cache). Replacement in a bin is often modeled by simple LRU policy, but modern caches rely on much more complex strategies [130]. For instance, Intel CPUs rely on a pseudo-LRU and 'random' evictions to reduce the cost of maintaining LRU [151]. Similarly, ARM CPUs implement a mix of LRU, FIFO, and random evictions [6].

Figure 3.2 illustrates the state of a 2-way cache, writing four arrays, one after the other. The first two arrays fit in the cache, and no conflict occurs (cache at t_0). When writing the third array, the cache needs to evict data. In strict LRU order, the cache would evict the first array and replace it with the third, but in reality the cache evicts data from both the first and the

second array (cache at t_1). When the fourth array is written, data from the three previously written arrays is evicted (cache at t_2).

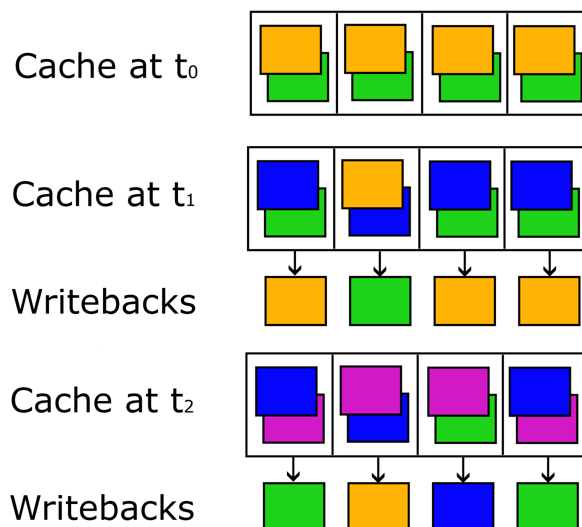

FIGURE 3.2: Content of a 2-way cache after writing four arrays, one after the other. Each array is presented in a different color. Because the cache evicts data in 'random' order, a given array may be partially evicted multiple times.

The problem of the random order of evictions is worsened when executing multiple threads on multiple cores. The interleaving of the memory accesses performed by the threads results in seemingly random memory accesses at the Last Level Cache (LLC) and decreases the likelihood of the cache evicting data sequentially.

Impact Non-sequential evictions are problematic when the size of the write unit of the cached medium is different from the CPU's cache line size.

For instance, on machine A, the Intel CPUs evict 64B cache lines, but the internal granularity of Optane Persistent memory is 256B. In the example of Figure 3.2, the first writeback at t_1 flushes four cache lines (for a total of 256B) to persistent memory, but internally the device writes $2 \times 256B$: 256B for the 3 cache lines of the yellow array and 256B for the 1 cache line of the blue array, resulting in a $2\times$ write amplification. The second writeback at t_2 results in a $3\times$ write amplification.

Guiding the cache. Instead of relying on the hardware-defined order of evictions, it is possible to use *clean* pre-stores to approximate sequential cache eviction. For instance, in the example of Figure 3.2, it is possible to ask the CPU to clean the first array (yellow cache lines) before reading the third array (blue cache lines). The first array is then very likely written to memory sequentially, and the reading of the third array causes no write amplification.

Performance Listing 2 presents a simple example to evaluate the impact of pre-stores on performance. Multiple threads write elements of an array in random order. The elements are then re-read to compute a total sum. We vary the size of the elements from 64B (simulating a succession of small random writes) to 4KB (simulating a succession of sequential writes). We evaluate the impact of pre-storing the elements just after their initialization.

Listing 2 Simple example to illustrate the impact of pre-storing data.

```
1 parallel_for(...) {
2   size_t idx = rand() % nb_elements;
3   memcpy(&elts[idx], ..., <sizeof elt>);
4   prestore(&elts[idx], <sizeof elt>, clean);
5   total += elt[idx].field;
6 }
```

Figure 3.3(a) presents the performance improvement brought about by pre-stores, and Figure 3.3(b) shows the write amplification when running Listing 2 in persistent memory. Adding pre-stores results in reduced write amplification and in up to a $3\times$ performance improvement.

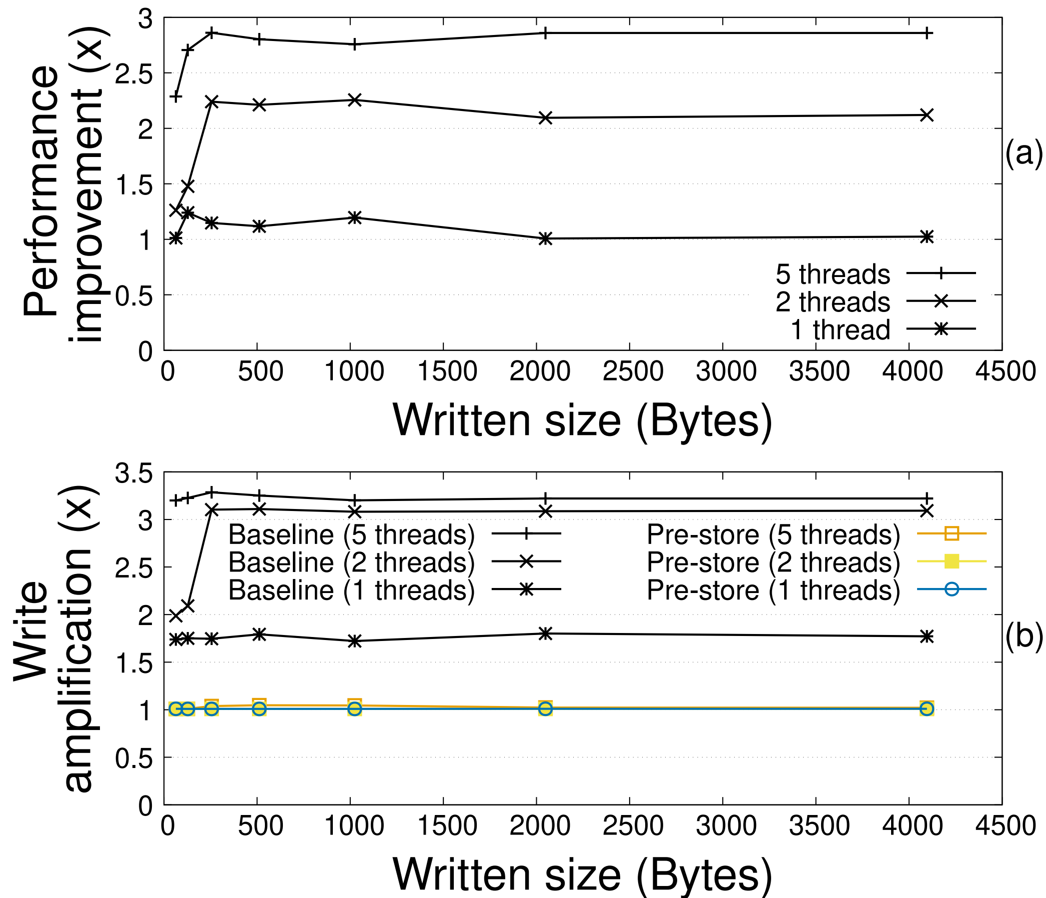

FIGURE 3.3: Machine A. (a) Improvement brought about by activating the *cleaning* pre-store call in Listing 2, varying the element size. (b) Write amplification with and without *cleaning*.

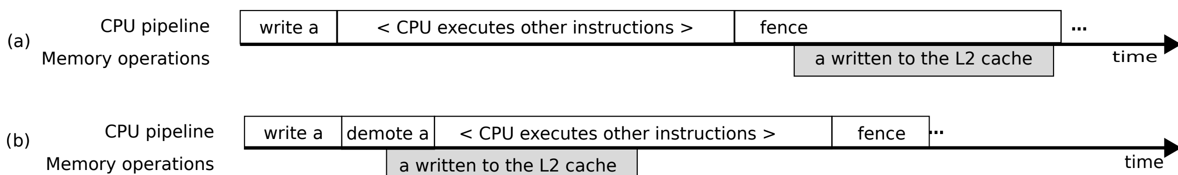

FIGURE 3.4: Demoting data forces the CPU to write the data to its caches. Without the demote instruction, the CPU can keep the modified a private, until it is forced to commit the change, causing delays in the execution of the CPU pipeline.

We measure the write amplification by comparing the number of 64B cache lines evicted from the cache to the amount of data actually written (both numbers are collected using the

`ipmctl` [31] tool). If it were the case that the cache was evicting data in the order it was written, pre-storing the elements would have no impact on performance when using large item sizes (e.g., with 4KB items, the cache would evict data in 4KB sequential chunks and would maximize PMem bandwidth). When using 1 thread, without pre-storing, however, the random order of evictions causes a 180% write amplification in persistent memory (every 64B cache line writeback results in 115B written in persistent memory). When using two or more threads, the write amplification grows to 330% (every 64B writeback results in 211B written in persistent memory), close to the maximum 400% write amplification for persistent memory. Pre-stores eliminate write amplification entirely.

The impact of eliminating write amplification on performance depends on the contention on the cached medium. With a single thread, the persistent memory is far from saturated, and the internal write amplification does not impact performance. With more than 2 threads, the write amplification limits the available bandwidth, and pre-stores improve performance by $2.2\times$ (two threads), and up to $3\times$ (five threads).

Summary It is possible to direct the cache to write dirty data sequentially to memory. Sequentiality improves performance, especially when the cache line size of the CPU and the internal write granularity of the cached device differ.

3.4.2 Problem #2 - Delayed cache operations (on Machine-B)

We explain why delayed cache operations can negatively impact performance.

Context When writing data, CPUs are allowed to keep the changes private, as long as the changes do not break the memory ordering constraints of the architecture. Because cache coherence operations are expensive, CPUs tend to keep modifications private and only advertise them when they run out of private buffer space or when they are forced to by the memory model.

Impact Figure 3.4(a) illustrates the impact of keeping modifications private. The CPU executes a write instruction, followed by other instructions, followed by a memory fence. The

reads and write prior to the fence must occur before the reads and writes placed after the fence. For brevity, in this chapter we refer to all instructions that implement the fence semantics simply as 'fences'. Other such instructions include, for instance, atomic instructions.

A consequence of the fence is that the CPU has to make all prior writes public before executing any other memory instruction. If the CPU has private modifications, publicizing these modifications happens 'at the last minute', while executing the fence, and stalls the CPU pipeline.

Guiding the cache. It is possible to force CPUs to write data to the cache, without stalling, using cache demoting instructions (*demote* operation of pre-store). Figure 3.4(b) illustrates the use of a demote pre-store. By issuing the pre-store, the application demotes the data, which triggers the data to be written to the cache in the background.

Impact on performance Listing 3 presents the pseudo-code of the example we use to assess the performance impact of delayed write operations. The example writes a cache line (128B on Machine B), demotes it, and then performs a configurable number of reads to its L1 cache. These operations are followed by a fence. We repeat the write-prestore-read-fence sequence 10 million times and measure the total runtime.

Listing 3 Pseudo-code to illustrate the performance impact of writes before a fence.

```
1 while(...) {
2     size_t idx = rand() % num_elements;
3     memset(&array[idx], ..., 128);
4     prestore(&array[idx], 128, demote);
5     for(int i = 0; i < n; i++)
6        read(&L1_data[i]);
7     fence(); // could also be an atomic op
8 }
```

Figure 3.5 presents the impact of the demotion on performance. We observe that demoting data is up to 65% faster than not demoting data on machine B.

The impact of the demotion depends on the ability of the CPU to overlap the demotion with other computations. If no reads are performed (left side of the graphs of Figure 3.5), the

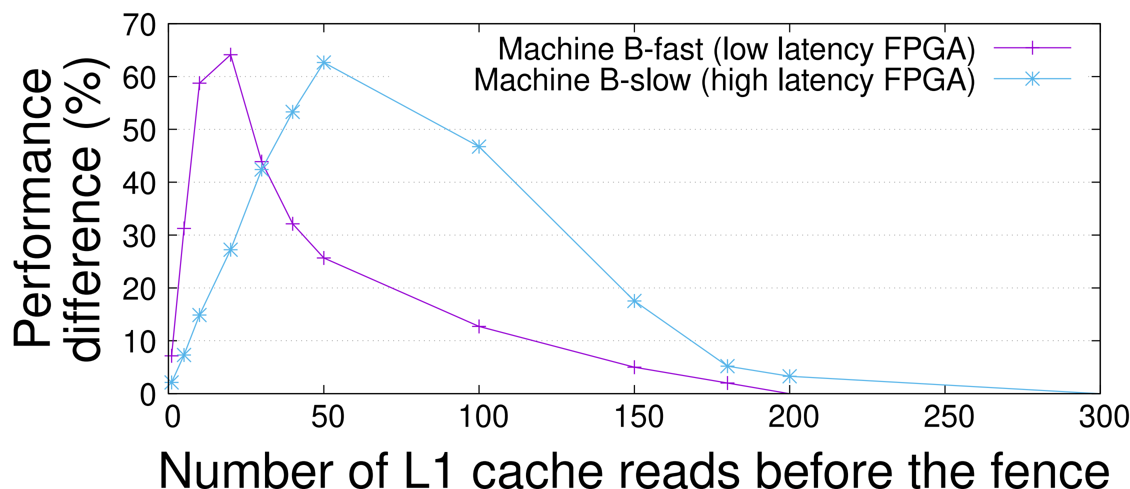

FIGURE 3.5: Machine B executing Listing 3. Relative performance improvement brought by demoting the dirty data before the fence. Demotion improves performance by up to 65%. The higher the latency of the cached device, the larger the window of time during which demotion improves performance.

fence happens just after the demotion order, so demotion cannot be overlapped with other operations. In that case, demotion provides no performance gain.

Slightly more counter-intuitively, the demotion operation also provides no performance gain when reading too much data from the L1 cache (right side of Figure 3.5). In this case, the runtime of the benchmark becomes dominated by the reads, and therefore the percentage benefits asymptotically go to 0.

Surprisingly, the impact on performance is highly dependent on the latency of the FPGA. The fast FPGA is most impacted by writes shortly followed by a fence, while the slow FPGA is most impacted by writes further ahead of the fence. The performance difference is explained as follows: when the cache receives the order to make a write visible, it needs to acquire the cache line in exclusive mode, and it needs to read the full cache line prior to updating it. Both operations are impacted by the latency of the FPGA:

- The time to read the cache line is obviously impacted by the latency of the FPGA.
- Acquiring the exclusive right to the cache line is also impacted by the latency of the FPGA. In many modern cache implementations, the cache directory is located on the cached device, instead of being stored in the cache itself. Intel CPUs, for instance,

store their directory in DRAM/PMem. Similarly, the ARM core maintains the status of the cached FPGA memory in the FPGA rather than in its cache. Every cache line status change thus requires accessing the FPGA.

So, the higher the latency of the FPGA, the larger the time needed to read a cache line and to update the cache directory.

Summary The delayed propagation of writes impacts the performance of applications that use fences or other instructions that enforce memory ordering (atomic operations, etc.). The impact depends on the latency of the cached device but, as CPUs are expected to cache an increasingly wider range of devices, it is likely to impact a wide range of workloads in the future.

3.5 Potential pitfalls of pre-stores

In the previous sections, we have shown that the lack of sequentiality in CPU evictions and the delayed propagation of write operations can have a high impact on performance. We have seen that the cache behavior can be controlled using pre-stores, leading to performance improvements.

When used correctly, pre-stores improve the management of the cache with little overhead. For instance, in the common case, *cleaning* a cache line simply enqueues a cache line in the write combining buffers of the CPU, which takes on average 1 cycle on our machines.

Inappropriate uses of pre-store Care must, however, be taken with the use of pre-stores to achieve the desired performance benefits. For instance, Listing 4 presents a slight variation of Listing 2 that constantly rewrites the same cache line instead of writing random elements of an array. In this microbenchmark, *cleaning* the cache line results in unnecessary writebacks to memory – without the pre-store, the data would just be overwritten in the cache. In such an extreme example, pre-stores result in a $75\times$ slowdown – an unsurprising result, equivalent to the ratio between the latency of writing to memory vs. writing to the cache.

Listing 4 Pseudo-code used to illustrate the overhead of pre-storing a frequently rewritten cache line.

```
1 char data[CACHE_LINE_SIZE];
2 while(...) {
3     memset(data, ..., CACHE_LINE_SIZE);
4     prestore(data, CACHE_LINE_SIZE, clean);
5 }
```

Skipping the cache In Listing 2 we have shown that cleaning is a simple and efficient way to avoid write amplification. We now consider a slight variation of Listing 2 in which Line 5 (the summation) is removed. In this case, skipping the cache is more appropriate because the data is no longer re-read, and therefore there is no reason for it to remain in the cache. Skipping the cache also directs the CPU to write data sequentially, and thereby also eliminates write amplification. Vice versa, in the original version of Listing 2, skipping is $2\times$ slower than cleaning, when using small elements, because skipping causes `elts[idx].field` to be read from memory instead of being read from the cache.

In large code bases, understanding where to add pre-stores by looking at application code can be time-consuming and error-prone. For instance, in the example of Listing 4, the rewriting of the cache line is obvious, but in more realistic code, it may appear in a different function, making it less visible to the developer. To address this problem, we have developed a tool called Dirtbuster. Dirtbuster analyzes the memory access patterns of an application. Based on that, it identifies the specific scenarios and locations where inserting pre-stores or skipping is beneficial. In the case of pre-store, Dirtbuster also suggests which type of pre-store to use.

3.6 DirtBuster: design and implementation of a tool for pre-stores

In this section, we present the design and implementation of DirtBuster. The goal of DirtBuster is to help developers find the code locations that could benefit from the addition of pre-stores.

3.6.1 Overview

As seen in Section 3.4, pre-stores improve the performance of sequential writes and of writes followed by memory ordering constraints. In order to find code sections that match these patterns, DirtBuster relies on dynamic analysis. Figure 3.6 illustrates the process.

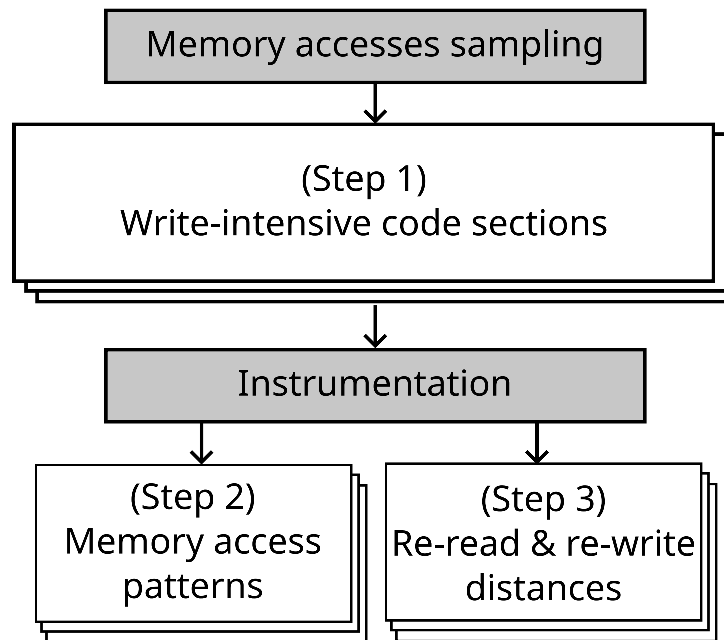

FIGURE 3.6: DirtBuster relies both on sampling and binary instrumentation. Sampling allows us to find the write-intensive functions, and binary instrumentation allows us to analyze the access patterns of these functions (sequential writes, writes before a fence, and re-read and re-write distances).

Step 1 The first step of DirtBuster consists of finding the write-heavy code sections of an application. This information is needed to know which functions to patch, and where to add pre-stores in the function. To get this information, DirtBuster relies on sampling memory accesses using performance counters.

Step 2 Pre-stores improve performance when the application performs sequential writes or when writes are followed by memory fences. We empirically found that the code surrounding write-intensive regions tends to be spread out in many files and libraries. For instance, atomic instructions, which also impose memory ordering constraints, tend to be called from external libraries (e.g., the atomic instructions of locks are generally called from the `pthread` library).

Such external dependencies made statical analysis of the code impractical. DirtBuster thus analyzes the binary of the application and of its libraries.

DirtBuster uses the Intel PIN tool [36] to log all reads and writes performed by the write-intensive functions, and to log all calls to memory fences. The logs are analyzed to check if the code writes data sequentially or if writes are followed by fences.

Step 3 For every memory region that is either written sequentially or written before a fence, DirtBuster helps developers to choose between *demoting* the data, *cleaning* the data or *skipping* the cache. The choice is based on the frequency at which the data is re-read or re-written. For instance, if the data is frequently rewritten, *cleaning* or *skipping* the cache would result in unecessary writes to memory (instead of simply being overwritten in the cache, the data would be pushed to memory every time).

To that end, we compute the re-write and re-read distances of every cache line. The re-write distance is defined as the number of assembly instructions between two consecutive writes to the same cache line. Similarly, the re-read distance is defined as the average number of assembly instructions executed between a read from a cache line and the preceding write to that cache line.

Intended usage DirtBuster is meant to be executed offline, as an optimization pass before re-leasing performance-critical applications. As a consequence, we did not focus on minimizing the overhead of the binary instrumentation pass. If an application can only be profiled and tested in performance-critical scenarios, it is possible to skip steps 2-3. In that case, it is up to the developer to understand if the write-intensive functions write data sequentially and if the data is re-used.

Using both sampling and binary instrumentation The use of both sampling (in step 1) and binary instrumentation (in steps 2 and 3) may seem redundant, but we found it useful in practice.

Sampling memory accesses is useful in step 1 because it allows understanding the behavior of an application with limited overhead (less than 1% in practice). The write-intensive

functions found using sampling are thus likely to be the ones that impact performance the most. However, sampling is too imprecise to be used in steps 2-3 to understand the memory access patterns of an application (e.g., sampling one memory access every 10K instructions is too coarse grain to detect sequential strides or to compute re-read or re-write distances accurately).

Conversely, binary instrumentation is too intrusive to be used in step 1: the instrumentation has a large overhead (up to $25\times$ slowdown in practice), and PIN's instrumentation causes cache thrashing, which may cause some cache-friendly functions to wrongly appear as memory-intensive. However, binary instrumentation allows gathering an exact view of all reads and writes performed by an application, which is necessary in steps 2-3 to understand the memory access patterns of the application.

3.6.2 Implementation

In this section, we present the implementation of DirtBuster, following the conceptual steps of Figure 3.6.

3.6.2.1 Step 1: Detecting write-intensive functions

DirtBuster samples memory accesses to find the functions that write data. Finding the functions is useful to know where to add the pre-store instructions.

DirtBuster relies on `perf` to sample the loads and stores performed by an application. DirtBuster gathers the time of all loads and stores, their instruction pointer (IP), and a callchain. The IPs are then grouped by functions to infer the most write-intensive functions. DirtBuster also groups the IPs of the callchains, to infer the most common paths that lead to these functions. Indeed, writes often happen in generic library functions (e.g., `memcpy`), and knowing the callchains that lead to these functions allows patching the relevant application code.

3.6.2.2 Step 2: Analyzing memory access patterns

In a second step, DirtBuster infers the memory access patterns of the write-intensive functions using binary instrumentation. DirtBuster instruments binaries using Intel PIN [36] to record all reads and writes performed by the write-intensive functions found in the previous step.

Detecting sequential writes A naive approach to check if a program writes data sequentially is by checking each write operation to determine if it targets the same cache line as the previous write or an adjacent cache line. However, this approach is insufficient for applications that write temporary values in between sequential writes, or for applications that interleave sequential writes to multiple objects.

To avoid such problems, DirtBuster keeps tracks of multiple 'sequentiality contexts'. A 'sequentiality context' is a record of a memory region (range of virtual address) and the location of the last write within that region. When a write is performed, DirtBuster checks if it is adjacent to the last write performed in any 'context'. If a context is found, its metadata is updated, otherwise a new context is created.

We currently do not impose a limit on the number of contexts tracked by DirtBuster. In practice, we found that the write-intensive functions perform sequntial writes on only a few objects (e.g., a few large arrays).

At the end of the execution, for every write-intensive function, DirtBuster reports the percentage of the writes that happen in sequential contexts and the size of the contexts. For instance, DirtBuster may show that a `functionX` performs 50% of its writes in sequential contexts, and that 80% of the sequential writes are to regions of size 1KB and that the remaining 20% of the sequential writes are to regions of size 16KB.

Detecting memory ordering constraints To detect memory ordering constraints, DirtBuster computes the minimum number of instructions between the writes performed by the write-intensive functions and the next instruction with fence semantics. Instruction with fence semantics comprise memory fence instructions (e.g., `mfence`, `sfence`, ...) and the atomic

instructions that force the CPU to order memory accesses (e.g., `cmpxchg` orders reads and writes).

3.6.2.3 Step 3: Re-read and re-write distance

DirtBuster computes the re-read and re-write distance of every cache line accessed by the write-intensive functions. To that end, DirtBuster maintains a counter of the number of executed instructions. For every monitored sequential context and for every cache line written before a fence, DirtBuster stores the value of the counter at the latest recorded read and at the latest recorded write. The information is currently stored in a B-Tree.

Whenever a monitored context is re-written, its re-write distance is updated. The only exception is that, to prevent categorizing sequential writes as multiple rewritings of the same context, DirtBuster updates the rewrite distance only when a write breaks a streak of sequential accesses. We compute the re-write distance as the average counter value (average number of instructions) between two subsequent writes to the same cache line. A similar computation is done for the re-read distance.

Guiding developers When a write intensive function writes data sequentially, or when writes are followed by fences, DirtBuster proposes to use pre-store. DirtBurster reports the name of the function and the line(s) of code that perform writes. From the line(s) of code that perform writes, it is usually obvious to infer which variables are written, and so which variables to pre-store.

If the data is re-written, DirtBuster suggests inserting a *demote* pre-store because it allows making the data visible to the other CPUs before hitting the fence, but keeps the data in the cache to speed up the future re-writes. If the data is just re-read, DirtBuster suggests inserting a *clean* pre-store after writing the data. The *clean* directs the CPU to initiate a writeback of the dirty data, but keeps it in the cache to speed up future re-reads. If the data is not re-read nor re-written, DirtBuster suggests to *skip* the cache using non-temporal stores. If non-temporal stores are hard to implement, a developer can choose to *clean* the cache instead.

The analysis performed by DirtBuster is independent of the machine architecture, but the performance benefits of following Dirtbuster's recommendations in terms of pre-stores depends highly on the architecture's characteristics. For instance, on machine A, with the strong x86 memory model, little gain is to be expected from following the recommendations in terms of writes before a fence (the memory model forces the CPU to make writes visible in order, so writes are rarely kept private in the CPU buffers). Likewise, on machine B, with the cache line and the memory unit having the same size, no benefit is to be expected from the recommendations in terms of sequential writebacks.

3.7 Evaluation

In this section, we aim at answering the following questions:

- Are pre-stores useful on real applications, and how big is the performance gain?
- Is DirtBuster giving the right recommendations? What is the effort of patching applications?
- Is there any downside in using pre-stores?

3.7.1 Setup

Machines A and B are described in Section 3.3. Table 3.2 presents the applications used in the evaluation. They contain a subset of the Phoronix benchmark suite[1] [108], two key-value stores; a subset of the NAS benchmarks [10], a compilation of HPC workloads, and X9 [55], a message passing library. Since pre-stores are most useful when writing data, we focus our evaluation on write-intensive applications. In particular,

- Some applications spend less than 10% of their time issuing store instructions (we used `perf` to get this information). Adding pre-stores to these applications would have no effect. We did not instrument these applications further.

[1] The Phoronix benchmark suite contains 497 test suites, many of which contain multiple applications. For practical reasons, we only executed a subset.

- We use DirtBuster to instrument the remaining applications. When DirtBuster recommends adding pre-stores, we patch the application according to DirtBuster's guidance. To further illustrate the usefulness of DirtBuster guidance, we also patch a few applications in which DirtBuster did not find any interesting pattern and show the overheads of pre-stores in such cases.

We focus the evaluation on TensorFlow [107], a machine learning framework, the 9 applications of the NAS benchmark suite and two variants of a key-value stores, one using a CLHT [39] hashmap as index, the other using a Masstree [106] index, and the X9 [55] message passing library.

3.7.2 Improving sequentiality (on Machine A)

We first evaluate applications in which DirtBuster detects sequential writes.

3.7.2.1 Machine learning workload

We benchmark TensorFlow, training a CNN. We execute the default ML training workload of the `pts/tensorflow` benchmark of the Phoronix benchmark suite [108]. We use the default input parameters and vary the batch size between 0 and 250. The batch size determines the number of images used simultaneously in an iteration of training; the higher the batch size, the larger the tensors used during training.

Step 1: finding the function to patch The sampling phase of DirtBuster shows that most writes to memory are performed in a templated function of the Eigen tensor library [59]: `Eigen::TensorEvaluator<...<op>...>::run()`. The `<op>` template parameter determines the operation performed on the tensor. For instance `Eigen::internal::scalar_sum_op<>` sums two tensors and `Eigen::internal::scalar_product_op<>` performs a scalar product of two tensors. Collectively, all the templated versions of the function account for 50% of the writes to memory when the benchmark is launched with a small batch size (0-50) and for 30% with large batch sizes (50+). DirtBuster indicates that most writes are performed in the loop

TABLE 3.2: Applications used in the evaluation. Some applications are not write-intensive, and would not benefit from pre-stores. For write-intensive applications, we used DirtBuster to detect memory access patterns.

	Write-Intensive	Sequential writes	Writes before fence
pytorch	✗		
numpy	✗		
lzma	✗		
c-ray	✗		
arrayfire	✗		
build-kernel	✗		
build-gcc	✗		
gzip	✗		
go-bench	✗		
rust-prime	✗		
TensorFlow	✓	✓	
X9	✓	✓	✓
Key-value Stores			
Masstree	✓	✓	✓
CLHT	✓	✓	✓
NAS benchmarks			
UA	✓	✓	
LU	✗		
EP	✗		
IS	✓	✗	
FT	✓	✓	
CG	✗		
BT	✓	✓	
MG	✓	✓	
SP	✓	✓	

that calls the inlined `evalPacket` function, which evaluates the operation and writes the result to a tensor (see Listing 5).

Step 2: understanding the memory access patterns Understanding the code of Eigen library of TensorFlow is beyond the scope of this chapter, but we used the binary instrumentation provided by DirtBuster to understand the memory behavior of the unrolled loop:

```
Eigen::TensorEvaluator<...<op>...>::run()
Location:  <...>/TensorExecutor.h line 272
Perc.  Seq.  Writes:  50%
```

```
Size:  16.2MB - 10% - re-read inf - re-write inf
...
Size:  240B - 60% - re-read 2 - re-write inf  Pre-store choice:
clean
```

Listing 5 Pseudo-code of a section of Eigen library used by TensorFlow to do tensor computations.

```
1  void Eigen::TensorEvaluator<...>::run(...) {
2    ...
3    for (; i <= last_chunk_offset; i += ...) {
4      evaluator.evalPacket(i + 0 * PacketSize);
5      evaluator.evalPacket(i + 1 * PacketSize);
6      evaluator.evalPacket(i + 2 * PacketSize);
7      evaluator.evalPacket(i + 3 * PacketSize);
8      prestore(&evaluator.data()[i], 64, clean);
9    }
10   ...
11 }
```

The `Eigen::internal::scalar_sum_op<>` function is templated and used to perform tensor operations on tensors of various sizes. Fifty percent of the writes performed by the function occur in sequential contexts. Of these, ten percent involve 16.2MB tensors, while sixty percent involve 240B tensors. For the 16.2MB tensors, the re-read and re-write distances are effectively 'infinite,' meaning the tensor is neither re-read nor re-written. In contrast, the smaller tensors are re-read almost immediately, with an average of just two instructions between the write and the re-read.

Step 3: choosing the correct type of pre-store Because the function is templated, the same code is used to operate on the 16.2MB tensors as on the 240B tensors. Since the re-read distance for the 240B tensor is very small, DirtBuster suggests using a *clean* pre-store. This choice would unlikely be the default choice of a developer, because nothing in the code suggests that the data is reused, and the manually unrolled loop can easily be modified to use non-temporal instructions. Without DirtBuster a developer would likely choose to *skip* the cache.

Performance To check the correctness of the recommendations of DirtBuster, we evaluate the performance improvements of both *cleaning* and *skipping*. *Cleaning* required adding a single pre-store line, while *skipping* required modifying the `evalPacket` function to use non-temporal instructions. The performance improvements on Machine A are presented in Figure 3.7. The performance improvement depends on the batch size used for the training. With small batch sizes, the performance improvement of *cleaning* is up to 47% (batch size 1), dropping to 20% with larger batch sizes. As DirtBuster suggested, *cleaning* the cache is the right optimization, and using non-temporal writes (*skipping* the cache) reduces performance by 20%.

Such large performance variations after patching a single function of TensorFlow may seem surprising, but, as mentioned earlier, the function represents half of the writes to memory for small batch sizes and a third for larger batch sizes. Without pre-storing, TensorFlow is memory-bound, limited by the speed at which the CPU evicts data. As illustrated in Figure 3.8, without *cleaning*, write amplification is $3.7\times$, while with *cleaning*, it drops to $2.7\times$, explaining the performance gains.

In these experiments, we only patched a single function of TensorFlow, which explains why pre-storing does not fully eliminate write amplification. DirtBuster detects other write-intensive functions, but indicates that they do not write data sequentially, and therefore we did not modify them to add pre-stores. Trying to do so, unsurprisingly, had no effect on performance. While pre-stores have no noticeable overhead, in this case they have no effect on the sequentiality of evictions and therefore offer no performance improvement.

Using non-temporal stores results in a performance loss because the newly written values depend on previously written values. The `evalPacket` function starts by loading a previously written packet, computes a new value and then stores the new value (i.e., a pattern similar to `a[x] = f(a[x-4*PacketSize])`). Profiling shows that `skipping` the cache doubles the time spent loading the value of the previously written packet.

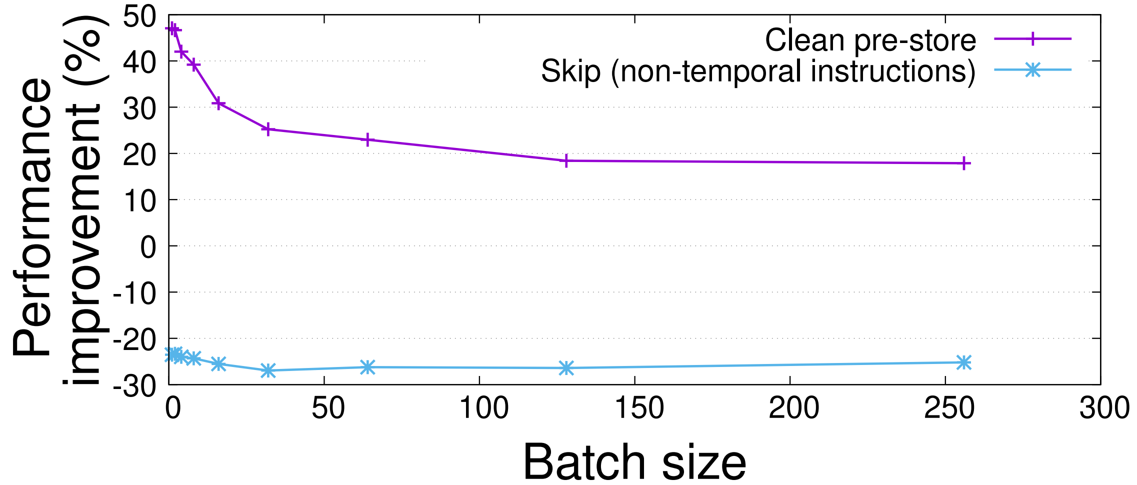

FIGURE 3.7: Machine A. TensorFlow. Performance improvement brought by pre-storing data. As advised by DirtBuster, cleaning the cache improves performance, while skipping the cache decreases performance.

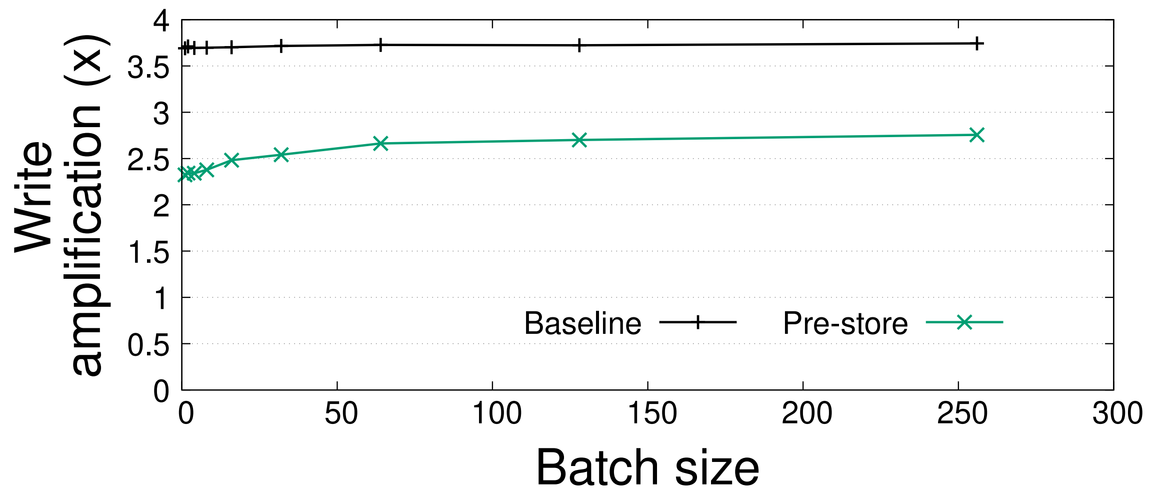

FIGURE 3.8: Machine A. TensorFlow. Adding a cleaning pre-store significantly reduces write amplification.

3.7.2.2 HPC workloads

We evaluate applications from the NAS benchmark suite. From all applications contained in the suite, only MG, FT, SP, UA and BT spend more than 10% of their time writing data.

Analysis of MG MG performs a multi-grid method on a sequence of meshes and is implemented as a succession of matrix multiplications. MG allocates 3 matrices, U, V and R. DirtBuster

detects that the `psinv` function writes the `U` matrix sequentially and that the `resid` function writes the R matrix sequentially. For the `resid` function, the output of DirtBuster is as follows:

```
Location: <...>/mg.f90 line 544
Perc. Seq. Writes: 100%
Size: 2.1MB- 100% - re-read 23.8K - re-write inf
Pre-store choice: clean
```

For the `psinv` function, the output looks as follows:

```
Location: <...>/mg.f90 line 614
Perc. Seq. Writes: 100%
Size: 2.1MB - 100% - re-read inf - re-write inf
Pre-store choice: skip
```

The data is written in chunks of 2.1MB. DirtBuster suggests *cleaning* the cache in the `resid` function because the data is re-read, and *skipping* the cache for the `psinv` function because the data is not re-read not re-rewritten (‘infinite’ number of instructions between two accesses to the same cache line). Unfortunately, Fortran offers no standard way to use non-temporal stores, so we chose the next best option, i.e., to *clean* the cache in both functions. Listing 6 presents the 1-line change to the `psinv` function. A similar change is made to the `resid` function. None of the authors had any prior knowledge of Fortran before changing the code of the NAS benchmarks, but DirtBuster helped pinpoint the exact matrix and code location that could benefit from pre-storing, and adding the pre-store instructions took less than an afternoon of work.

Analysis of FT, SP, UA and BT FT performs a Fast Fourier Transform. DirtBuster indicates that the `cffts1` function sequentially transfers results from a matrix `Y1` to a matrix `XOUT`. SP is a Scalar Penta-diagonal solver. DirtBuster detects that SP allocates dozens of matrices, but that a single matrix (`RHS`) accounts for most of the writes. The matrix is mostly written in the `compute_rhs` function and is rarely reused. Similarly to the change done in MG, we

Listing 6 Change made to the `psinv` function of MG.

```
1  subroutine psinv( r,u,n1,n2,n3,c,k)
2  !$omp parallel do ...
3  do i3=2,n3-1
4    do i2=2,n2-1
5      do i1=2,n1-1
6        u(i1,i2,i3) = u(i1,i2,i3)
7            + c(0) * r(i1,i2,i3) + ...
8      enddo
9      call prestore(loc(u(2,i2,i3)), ..., clean)
10   enddo
11 enddo
```

chose to *clean* the matrices after writing them. Similarly, UA and BT write matrices that we also chose to *clean*.

Performance Figure 3.9 summarizes the performance of the NAS benchmarks with pre-storing on Machine A. Pre-storing is up to 40% faster.

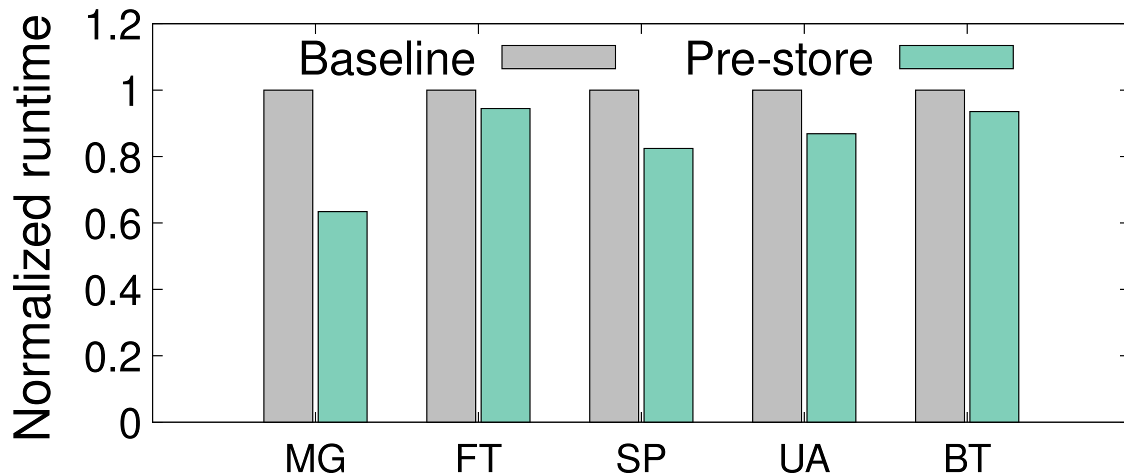

FIGURE 3.9: Machine A. Normalized runtime of the NAS benchmarks. Lower is better.

3.7.2.3 Key-value stores

In this section, we evaluate the performance of the CLHT and Masstree key-value stores, running a YCSB workload. We configure the key-value stores with 100 million keys and evaluate

value sizes ranging from 64B to 4KB. We inject load using 10 threads, the configuration that provides the highest throughput.

Analysis When executing YCSB, DirtBuster indicates that read-only or read-mostly workloads (YCSB B-D) do not benefit from pre-storing data. However, on YCSB A (50% GET, 50% PUT), DirtBuster suggests *skipping* the cache when executing a PUT query. DirtBuster indicates that the values inserted in the key-value store are rarely reused, that they are written sequentially, and that the writes are followed by a fence.

Performance We compare the performance of the unmodified code (baseline) against versions using pre-stores. In the first version, we used non-temporal stores to craft the values of the PUT function (suggestion of DirtBuster). In the second version, we clean the cache after crafting the value. Listing 7 presents the version of the PUT function of CLHT when cleaning the *cache*. *Skipping* the cache required rewriting the `craftValue` function).

Listing 7 Pseudo-code of PUT function of CLHT.

```
1 void ycsb_put(...) {
2     void *value = craftValue(...);
3     prestore(value, size, clean);
4     clht_put(..., value);
5 }
```

Figures 3.10 and 3.11 summarize the performance of CLHT and Masstree on Machine A, running YCSB A. Skipping the cache is up to $2.9\times$ faster than the baseline for CLHT and $2.5\times$ faster for Masstree.

As advised by DirtBuster, *skipping* the cache offers the best performance, but it also requires significant changes to the `craftValue` function. *Cleaning* only requires adding a single line of code and is still up to $2.3\times$ faster than the baseline for CLHT and up to $1.9\times$ faster for Masstree. In complex code bases, it is thus possible to simply *clean* the cache to achieve significant performance gains.

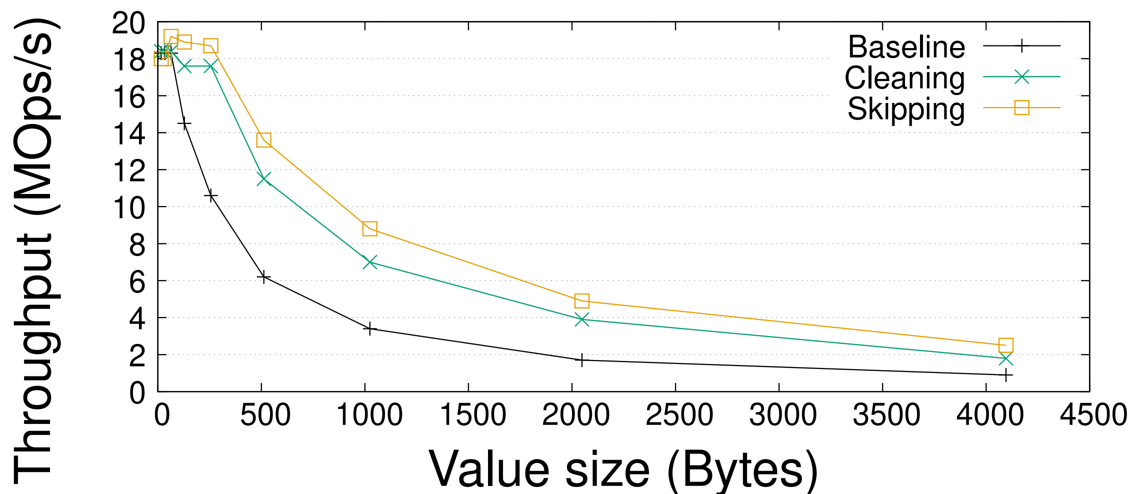

FIGURE 3.10: Machine A. CLHT performance, in requests per second. Higher is better.

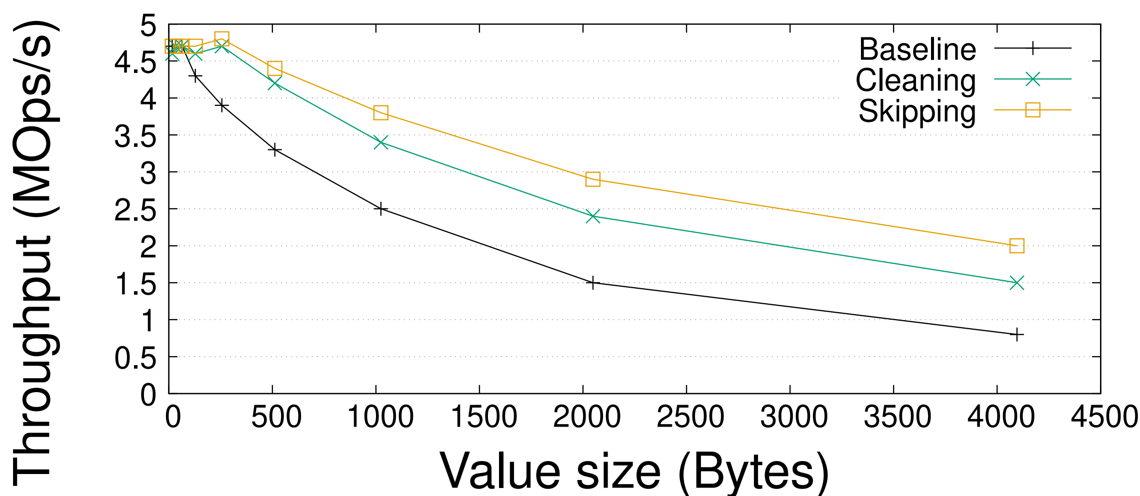

FIGURE 3.11: Machine A. Masstree performance, in requests per second. Higher is better.

Profiling shows that skipping and cleaning the cache reduce write amplification. When running YCSB with values larger than 256B, for the baseline every 64B evicted from the cache results in an average of 248B written to persistent memory ($3.8\times$ write amplification, see Figure 3.12). Skipping and cleaning both eliminate write amplification when YCSB is configured to use large values. Skipping outperforms cleaning because the crafted values no longer pollute the cache.

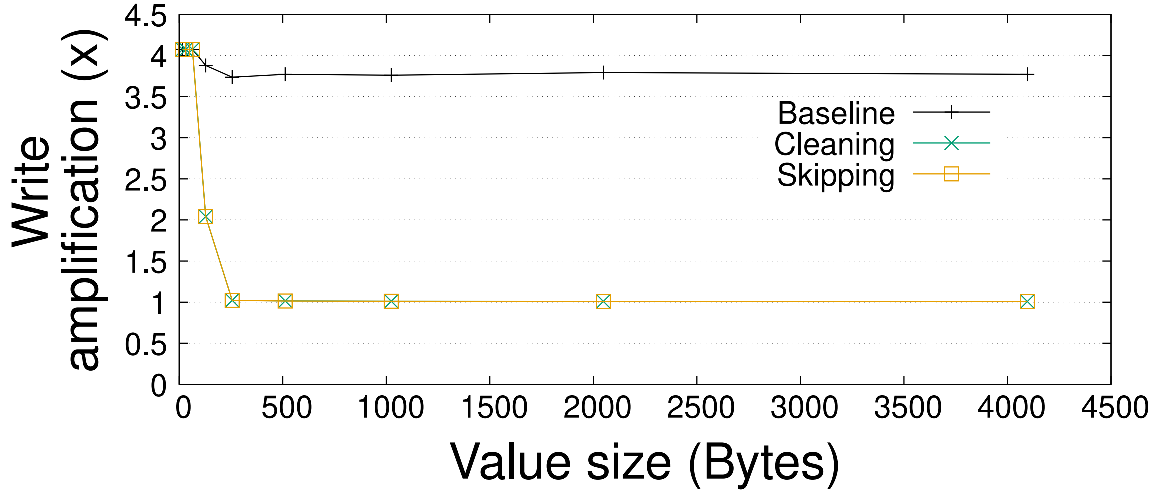

FIGURE 3.12: Machine A. Write amplification of CLHT executing YCSB A (lower is better).

As expected, the reduction in write amplification is the largest when the value size matches or exceeds the cache line size of persistent memory (256B). Pre-storing, however, improves performance as soon as the value size exceeds the cache line size of the CPU (64B). For instance, with 128B values, pre-storing halves the write amplification and improves performance by 21.4% for CLHT and 7.0% for Masstree.

3.7.3 Improving latency (on Machine B)

In this section we evaluate applications in which DirtBuster detects writes followed by memory ordering constraints.

3.7.3.1 Key-value stores

We run the YCSB workload presented in the previous section on machine B.

On machine A, pre-storing is useful because it increases the sequentiality of evictions. On machine B, the FPGA interleaves memory requests to multiple concurrent memory controllers, so the machine does not benefit from the increase in sequentiality. Pre-storing is, however, still useful because writes are followed by fences and atomic instructions. To measure the

performance of *cleaning*, we use the same patch as the one used for Machine A (see Listing 7). We also tried to port the code used to *skip* the cache on machine A, but could not implement a working version for machine B (non-temporal code is architecture-specific, and Arm CPUs do not offer standard libraries to implement non-temporal operations). As a consequence, we only present *cleaning* results in this section.

Figures 3.13 and 3.14 present the performance with 1KB values. Pre-storing is 52% faster on CLHT and 25% faster for Masstree. Pre-storing is most useful when machine B is configured to use the fast FPGA because the memory ordering instructions happen soon after writing data.

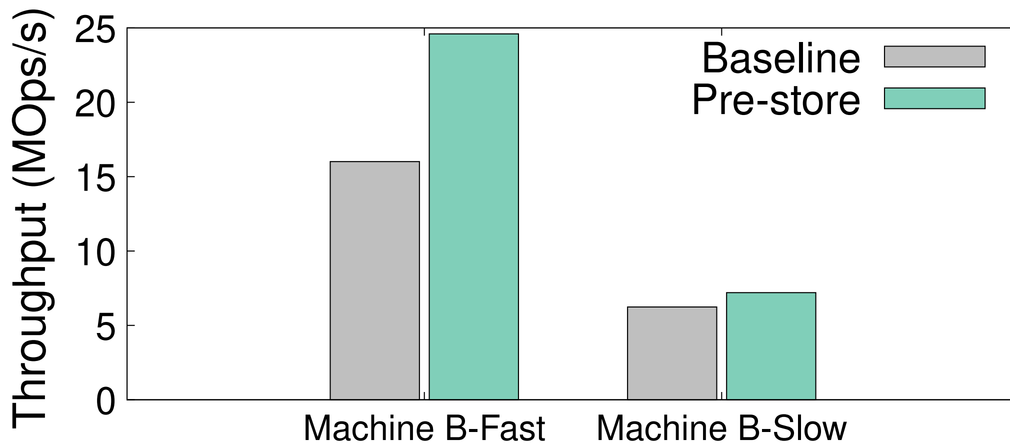

FIGURE 3.13: Performance of CLHT on Machine B-fast (FPGA configured with a low latency) and Machine-B-slow (high-latency FPGA).

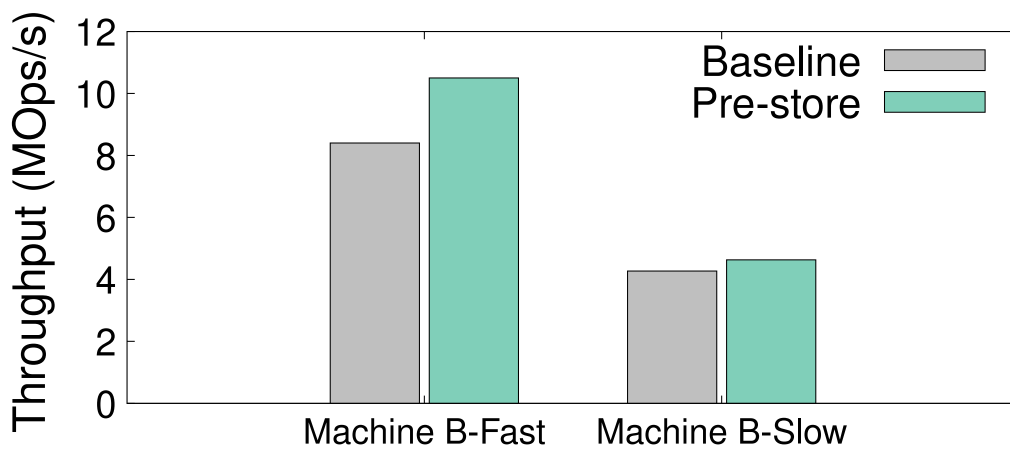

FIGURE 3.14: Performance of Masstree on Machine-B-fast and -slow.

When inserting an object, CLHT computes the hash of the object and then locks the bucket it belongs to. The atomic operations used in the lock have a fence semantics and force the CPU to make the crafted value visible to all the cores in the machine. Profiling shows that pre-storing allows making the value visible before reaching the atomic operation, reducing the time spent in the atomic instructions of the lock by 74%.

Similarly, to ensure correctness, Masstree uses version numbers for all of its objects. When reading or updating an object, Masstree checks the version number of the object and then re-checks the version number after manipulating the object, to detect possible concurrent changes. To enforce the correct order of the operations, Masstree uses fences. Listing 8 presents the logic of the code for the insertion of a new value in the KV. The fences are mandatory for correctness, but they may cause the CPU to stall if the crafted value has not been made visible to all the cores of the machine. Profiling shows that pre-storing the values halves the time spent in the first fence of `masstree::put`.

Listing 8 Pseudo-code of the actions performed while inserting a new value in Masstree. The fences ensure correctness, but impact the pipeline of the CPU.

```
1  void *masstree::put(...) {
2      while(...) { // tree traversal
3          v = node->readVersion();
4          if(isLocked(v))
5              goto restart;
6          fence();
7          ... // read or modify the node
8          fence();
9          if(node->versionChanged(v))
10             goto restart;
11     }
12 }
```

3.7.3.2 Message passing workload

We benchmark the X9 message passing library. We measure the latency of sending messages from a writer thread to a reader thread (see Listing 9). DirtBuster detects that the crafting of a message, in function `fill_msg` is followed by a compare-and-swap instruction in the `x9_write_to_inbox` function. DirtBuster also detects that the benchmark rewrites

the same message structures multiple times: X9 reuses the message structures to avoid the overheads of allocations on every message exchange. DirtBuster recommends to *demote* the messages after writing them.

We added a *demote* pre-store after `fill_msg`. The pre-store reduces the latency of sending a message by 62% on machine B-fast, and 40% on machine B-slow. Profiling shows that the pre-store reduces the time spent in the compare-and-swap. Without pre-store, the message is kept in private CPU buffers and is only made visible 'at the last minute', when executing the compare-and-swap. With pre-stores, the message is sent to the L2 cache asynchronously, before reaching the compare-and-swap instruction.

Listing 9 X9, pre-storing before sending a message.

```
1 void* producer_fn(...) {
2  fill_msg(&m[...]);
3  prestore(m[...], sizeof(msg), demote);
4  x9_write_to_inbox(inbox, sizeof(msg), m[...]);
5 }
```

3.7.4 Overhead of pre-stores

In this section, we analyse the overhead of pre-stores, when they are not needed. We distinguish pre-stores suggested by DirtBuster, on architectures that do not benefit from it, and pre-stores that we tried to add manually, without DirtBuster guidance.

3.7.4.1 Pre-stores suggested by DirtBuster

The NAS applications and TensorFlow only use a fraction of the available bandwidth of Machine B, and they do not use fences. As a consequence, pre-stores do not provide any benefit on machine B. We still evaluated the performance of the patched applications, cleaning data after writing the tensors and the matrices. Unsurprisingly, adding pre-stores does not have a positive impact on performance, but the maximum overhead was limited to 0.3%. In general, we could not find applications for which adding pre-stores at the locations recommended by DirtBuster added any significant overhead.

3.7.4.2 Incorrect manual use of pre-stores

For completeness, we tried to patch some functions and applications in which DirtBuster does not recommend using pre-stores.

FT from the NAS benchmark suite DirtBuster suggested adding pre-stores to the `cffts1` function of FT, which resulted in performance gains on machine A (see Section 3.7.2.2). When profiling the application with `linux perf`, we noticed that another function, called `fftz2`, is also write-intensive. While a manual reading of the code suggests that it is sequentially writing data, it is not immediately apparent that the written data is small enough to fit in the cache and is frequently re-read and re-written. *Cleaning* the cache in that function resulted in a $3\times$ slowdown. DirtBuster was able to detect both the length of the sequential writes and the distance between rewrites, and did not suggest using a pre-store.

IS from the NAS benchmark suite While manually profiling IS using `perf`, we observed that a single function, `rank`, is responsible for the majority of writes. Similar to the previous case, this function contains a small loop that might initially seem like a suitable candidate for pre-stores. However, the function actually writes small amounts of data in a seemingly random pattern. In this case, adding a pre-store has no effect (no performance gain, no overhead) because the written data is neither re-read nor re-written. DirtBuster detects the lack of sequentiality and does not suggest using a pre-store.

3.7.4.3 Manual intervention and edge cases

DirtBuster is able to suggest relatively precise locations to insert Pre-stores. However, the insertion should still be manually checked by programmers. For instance, in Listing 6, an optimal location to insert Pre-stores is outside the innermost loop, where the writes are occurring. Programmers can also insert Pre-stores inside the innermost loop, but with careful checks on granularity. Only after a suitable amount of writes has accumulated should Pre-stores be performed. Otherwise, improper use of Pre-stores results in redundant writes to memory and can have a negative impact on performance. For the cases of machine B, there might be even more lines of code between the location where writes are occurring and the

memory barrier, and there might be many locations suitable for Pre-stores. Generally, they should not be very far from the writes.

With the design of binary instrumentation and contexts, DirtBuster is able to detect most cases correctly. Pre-stores can distinguish objects or memory regions by sources of allocation such as malloc(). In cases where allocation tracking fails or results in high overhead, for example in Python where there are too many small allocations, DirtBuster also employs smart solutions such as filtering small allocations or grouping allocations or objects based on the code location of the allocations. In addition, since most objects of the same type usually fall into the same page, DirtBuster also uses page-based context as a fallback. It works well and has been validated on complex code bases such as TensorFlow.

Conclusion The examples above demonstrate that using DirtBuster is a valuable approach when optimizing memory write operations in applications. DirtBuster not only identifies scenarios where pre-stores (or skips) can enhance performance, but also refrains from recommending their use where they would not result in any gains or could even produce slowdowns.

3.8 Related Work

Directing the cache The idea of controlling hardware caches in software dates back from the 80s [119]. Most work since then has focused on helping the caches to prefetch data in order to hide the latency of memory, either by improving prediction capabilities in hardware [73, 115] or by directing caches with software prefetch instructions [103, 86, 111, 152, 121, 110, 73, 115]. More recently, the advent of persistent memory has brought back the necessity of cleaning cache lines to offer persistence guarantees. Most published work has focused on detecting incorrectly placed cleaning instructions to find persistence bugs [56, 99]. To hide the high latency of writing data to PMem [174, 123], Shin et al. [137] and Ribbon [153] have proposed to evict dirty data to persistent memory in the background. Xu et al. [161] have proposed to cache written data with higher priority to avoid unnecessary writebacks. In

DirtBuster, we propose the more general concept of a pre-store and show its usefulness to improve sequentiality and to reduce latency on weak memory architectures.

Software approaches to influence caching Multiple strategies have been proposed for maximizing the efficiency of CPU caches in software. Khan et al. [82] have proposed to partition the cache to avoid conflicts between read-mostly and write-mostly data. In general, the careful placement of pages in virtual and physical memory [81, 15, 90] has been used to reduce conflicts in the cache. Page coloring techniques partition the cache to avoid cache thrashing between users or applications [173, 18]. Scheduling techniques ensure that critical applications are co-scheduled with applications that do not thrash their CPU caches [141, 9, 178]. These techniques are orthogonal to the ones explored in this chapter.

Sequentiality The importance of sequentially accessing cached devices has been mentioned in previous work. Most work have focused on designing data structures that can be read and written in long sequential strides [78, 154]. While the code accesses data sequentially, no guarantee is provided at the hardware level. DirtBuster could be used to enforce that the CPU evicts data in sequential order.

Cleaning the cache In this chapter, we use cleaning instructions to improve performance. Cleaning the cache has also been used to reduce the likelihood of timing attacks [75].

The term 'pre-store' in the literature The term "pre-store" was used in the related work to refer to different concepts. Chen et al. [26] propose to add a non-cache coherent buffer on top of a cache. Data is 'pre-loaded' in the buffer but, because the buffer is non cache-coherent, it cannot contain entries that are later written. What the paper defines as a 'pre-store' is an instruction that tells the buffer to not 'pre-load' a given memory address in the buffer. Kim et al. [84] focus on the DRAM cache of SSDs. They propose to automatically evict dirty data from the DRAM cache to storage in the background, when the storage is idle (the background eviction scheme is referred to as "pre-storing"). Doing a similar strategy for the CPU cache would require hardware changes because only the CPU knows when its memory controller is idle. In our context, a "cache pre-store" is an assembly instruction that directs the CPU to move data down the memory hierarchy.

3.9 Conclusion

We have introduced the concept of pre-stores for moving data asynchronously down the memory hierarchy – the converse of prefetches that move data up in the memory hierarchy. We have demonstrated the benefits of pre-stores on two types of architectures: one in which the CPU cache line size is different from the write unit of the underlying memory and one in which updating the cache exhibits long latencies.

To assist the developer in inserting pre-stores in judicious program locations and in avoiding potential pitfalls in doing so, we have presented the Dirtbuster tool, a dynamic analysis tool that uses memory access sampling and instrumentation to discover code patterns suitable for insertion of pre-stores.

We have used Dirtbuster on a large number of benchmarks. We have demonstrated performance improvements of up to $2.3\times$ on some benchmarks, without incurring performance regression on any of them.

Hardware pre-stores. Similar to the hardware counterpart of software prefetch, we believe that it is possible to have hardware Pre-stores, and we hope that our work can inspire their development. A key initial problem we identified is that hardware such as CPUs has no knowledge of the type or speed of the physical memory mapped into its address space for the purpose of byte addressability in current heterogeneous architectures. Many of these devices perform suboptimally when hardware treats them as traditional DRAM. For example, some prefer writes to be sent in batches rather than randomly, while others prefer updates to be issued ahead of time to hide latency. Hence, as a first step, this information should be made available to hardware, and hardware should acknowledge it in its design. Moving beyond that, hardware could then speculate on the movement of data and perform writes before it is forced to do so.

Acknowledgements. We would like to thank our shepherd, Horst Schirmeier, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Australian Research Council Grant DP210101984. We thank the Enzian team

at ETH Zurich for access to the Enzian machines. We thank Shuaiwen Song and the members of the FSA lab for their guidance and feedback throughout the project.

CHAPTER 4

Opportunities With Emerging High Speed Unified Memory

Discussion: Pre-stores greatly improve application performance when using heterogeneous memory. However, these issues exist fundamentally because memory system performance does not match the compute tier. We believe that the memory system will evolve, and a driving force for this evolution is artificial intelligence.

One obvious observation is that AI demands a large amount of memory at once, along with a large amount of bandwidth, at an unprecedented level. Hence, it is wise to disaggregate dedicated memory from its compute device and use a global, shared memory pool instead to resolve the imbalance and costly data movement associated with dedicated memory. With that in mind, we explore opportunities in a system that is closest to the shape of a future memory architecture.

As the last chapter of my thesis, we study emerging computer memory design. There is little doubt that the future memory will be faster, improving overall performance while allowing more and more devices to collaborate efficiently. Through this chapter, we take a glance at the future and upcoming opportunities.

4.1 Introduction

Among all the most critical components of a computer system are CPU, GPU, and system memory. CPU (Central Processing Unit) is a general purpose compute device used for many tasks. Conversely, a GPU (Graphics Processing Unit) is a specialized processor designed to handle parallel computation tasks. They were originally for graphics, but now widely used for

data-parallel workloads. GPUs get divided into two types: (1) Integrated GPUs (iGPUs) and (2) Discrete GPUs (dGPUs). Almost all compute devices require memory. CPU has access to system memory, however, which memory backs GPU depends.

Definition of unified memory: For the purposes of this thesis, we define unified memory strictly as a memory architecture in which all devices have complete and equal access to the entire memory space, with no portion being exclusively reserved for a specific device. It is worth noting that unified memory is often associated with integrated GPUs (iGPUs). While we acknowledge this common association, we distinguish between iGPUs and unified memory in this work. This distinction is important because discrete GPUs (dGPUs) can also employ unified memory, and, conversely, some iGPUs may still retain exclusive access to a portion of video memory.

Current unified memory and iGPUs: iGPUs are built into the CPU's package or same chip. The CPU shares system memory with them, and they do not own any dedicated graphics memory. This can also be referred to as unified memory architecture. Previous studies [132, 104, 58, 67, 54] have shown that integrated GPUs can effectively accelerate or complement CPU workloads for a range of applications since they can access system memory in a direct fashion and this can benefit them with a reduced overhead in data movement. However, these platforms do rarely offer memory access that is truly unified or symmetric in actual practice. Many systems [47, 46, 131, 70, 53] require manual configuration of the memory split between CPU as well as GPU through setting VRAM/DRAM thresholds with other systems [57, 148, 3] impose important latency or bandwidth penalties whenever one device accesses memory allocated to the other. Underlying hardware constraints along with software-level asymmetries often weaken the supposed benefits of shared memory.

Furthermore, most iGPU-based systems face a pair of critical limitations: (1) shared memory bandwidth is typically backed by DDRx channels, as this bandwidth is often already saturated by the CPU, thus leaving insufficient bandwidth for GPU-intensive tasks [22, 100, 3, 146], as well as (2) the integrated GPU itself generally offers limited computational throughput compared to discrete GPUs, due to a smaller number of execution units, lower clock speeds, and smaller caches. The scalability and effectiveness of these iGPUs for bandwidth- or

compute-intensive workloads are in a collective way limited by all of these constraints despite their own theoretical advantage in terms of minimizing memory copy operations [67, 58, 3, 5].

dGPUs: dGPUs exist as separate hardware units. System memory is not shared with dGPUs, but they feature their own dedicated memory. They achieve better performance along with higher computational throughput in demanding graphics, parallel compute, or GPU-accelerated workloads. dGPUs offer greater compute capabilities than integrated counterparts, together with dedicated high-bandwidth memory like 16-48GB of GDDR6 or HBM with bandwidths exceeding 400GB/s, yield substantially higher throughput across a wide range regarding parallel workloads [168, 20, 140]. However, architectural trade-offs come along despite these performance advantages. The PCIe bus typically limit data transfer speed, ranging from 16 to 64GB/s, depending on generation as well as lane width. Then accessing host memory or offloading datasets creates latency with large bandwidth overhead [168, 175, 76]. The fixed size of VRAM on-board often is not able to fully accommodate datasets or models that are large, resulting in frequent memory copying [96, 168, 160, 142].

Whenever working sets are larger than VRAM capacity or require dynamic host-device communication, repeated data transfers can cost more than what the GPU gains computationally. Such instances reveal a critical PCIe transfer latency bottleneck thus limiting the effective dGPU-accelerated workload performance plus highlighting memory architectures needing reduced explicit data movement [168, 76, 160, 83, 79].

Future unified memory and iGPUs: Hardware vendors develop unified memory architectures blurring the boundary between iGPUs and dGPUs because customary iGPU architectures have limitations, also because demand for on-device processing grows, especially deploying large language models locally for privacy-sensitive applications. These platforms are emerging so both processors can fully access high-bandwidth memory, thus costly data transfers are unneeded and memory bottlenecks are reduced because heterogeneous computing was formerly constrained.

Latest unified memory systems typically offer aggregate bandwidths greatly beyond what a CPU or GPU can saturate independently, enabling improved concurrency and workload co-execution. Integrated GPUs now have made some substantial progress in their architectural efficiency. Newer generations also gained higher throughput in parallel, though they still fall short of fully matching the raw performance of high-end discrete GPUs, their performance is now comparable to that of mid-tier discrete GPUs, so they are increasingly viable for a broad range of compute-intensive workloads. Efficient unified-memory-based heterogeneous computing does have some new opportunities because of each of the advances, especially in the edge and client devices.

Applications: Extensive prior work has explored the use of both integrated and discrete GPUs to accelerate applications, achieving notable improvements in performance and efficiency. In this chapter, we revisit previous designs in the context of emerging unified memory architectures and modern iGPUs, with the goal of pushing the boundaries further. To explore the potential of unified memory architectures, we conduct a comprehensive empirical investigation of their performance characteristics under diverse memory and compute intensive workloads. Building on these insights, we provide designs and guildlines that help applications to leverage both unified memory and the improved computational capabilities of modern integrated GPUs.

The contributions of this chapter are as follows:

Section 4.2 includes the comparison of unified memory to classical PCI-E and discrete graphics card architectures, as well as accounting for the large, seldomly utilised available system bandwidth. Motivated by these observations, in Section 4.3 we introduce, evaluate and optimize a former hash table which was only available on the GPU, taking advantage of the good latency and bandwidth characteristics it provides. Finally, we evaluate analytics query from database systems that run directly on discrete GPUs and show that unified memory system running query workload faster and more efficient in section 4.4.

4.2 Background

4.2.1 Discrete GPU

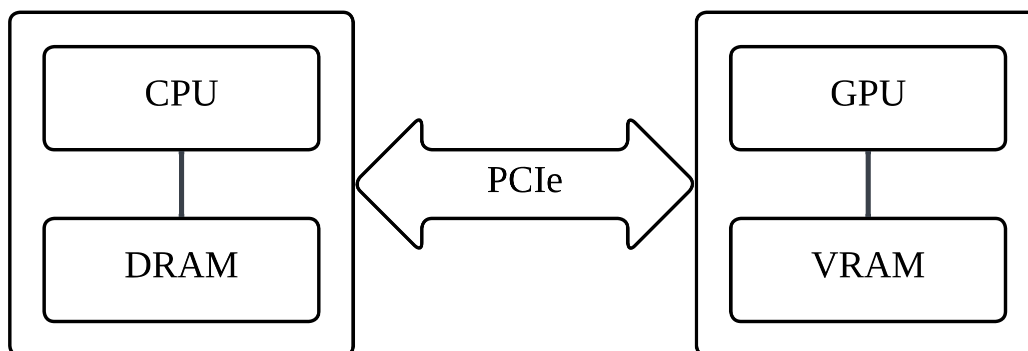

FIGURE 4.1: Illustration of a common machine with discrete GPU

Conventionally, in order to employ a discrete GPU for acceleration, a PCI Express (hereafter referred to as PCIe) socket attaches the GPU to the motherboard, as illustrated by figure 4.1. A discrete GPU connected thus cannot directly access data that resides in the host device's main memory (e.g. DRAM). As a result, a limited amount of dedicated memory is attached to the discrete GPU (hereafter referred to as VRAM). Data, however, must still be transferred from main memory and then to VRAM by way of PCIe before the discrete GPU can process it [175, 43, 30].

Discrete GPUs generally give increased computational power. However, for basic operations such as common data processing, the aforementioned PCIe data transfer cost has long been the primary bottleneck, instead of just compute power itself [168, 76, 160, 83, 79]. PCIe along with main memory as well as VRAM bandwidth are compared across a few generations by Table 4.1. Gigabytes of data are often transferred in tens of milliseconds since PCIe bandwidth is typically much lower than main memory and VRAM. At the current time, PCIe 4.0 is still the most widely deployed standard within servers, with PCIe 3.0 still being frequently used, despite the availability of PCIe 5.0 and later versions [44, 49, 134]. Upgrading to a higher

PCIe version that effectively doubles the transfer rate may provide a temporary performance improvement, yet comes at large expense because both the motherboard and the graphics card must be replaced. Consequently, a lot of the prior studies have concentrated in regards to alleviating that bottleneck. Improved data locality, reduced data movement, also the use of batching strategies help for achieving this [24, 116, 143, 21]. Still, these are software patches that could not fully overcome problems brought by hardware designs.

Version	Throughput (GB/s)	1GB transfer (ms)	Main memory (GB/s)	VRAM (GB/s)
1.0	4	250	25	86
2.0	8	125	43	150
3.0	15	66	51	177
4.0	31	32	204	484
5.0	63	15	250	700

TABLE 4.1: Speed of different PCIe versions compared to speed of mainstream main memory and VRAM, assuming all 16 PCIe lanes are used.

Generation	Year	Card Name	VRAM size (GB)	Price (USD)
Pascal	2016	Quadro P6000	24	5,999
Volta	2018	Quadro GV100	32	8,999
Turing	2018	Quadro RTX 8000	48	10,000
Ampere	2020	RTX A6000	48	4,650
Ada Lovelace	2023	RTX 6000	48	6,800
Blackwell	2025	RTX Pro 6000	96	8,565

TABLE 4.2: The price of each generation workstation cards and respective VRAM size from Nvidia.

Furthermore, one cannot overlook another critical limitation on available VRAM size. Increasing VRAM capacity usually incur great cost. The flagship workstation cards from Nvidia since the year 2016 are listed in Table 4.2 across all major releases, as well as respective price and VRAM size. Servers do not commonly have flagship discrete GPUs that offer larger VRAM capacities. Instead, most entry-level servers come with discrete GPUs with only 16GB of VRAM or less. Generally, the GPU costs as much as or is cheaper than the server's remaining components [96, 136]. As a consequence, to make use of the graphics cards shown in Table 4.2, the overall system cost often is more than double the price of the GPU alone.

The capacity of VRAM directly determines the volume of data a GPU can process in a single execution. When the dataset exceeds the available VRAM, several possible outcomes arise, all of which impose substantial performance penalties [45, 168]. One approach is to extract only the relevant data and transfer it to the GPU. However, this imposes significant additional workload on the host CPU. A second outcome is that the system reverts to CPU-based processing, which, without GPU acceleration, results in prohibitively slow performance for large datasets or indexes. A third option is provided by certain GPU kernel libraries, such as CUDA, which offer interfaces like cudaMallocManaged() to enable transparent data movement between host and device, thereby allowing processing to continue beyond the limits of available VRAM. In this case, DRAM functions as an extension of VRAM, with data being dynamically transferred from DRAM to VRAM when required, and evicted back when no longer needed. Compared to the scenario in which the dataset fits within VRAM, where transfers typically occur only at the beginning and end of computation, the overhead of continuous, on-demand transfers is prohibitively high.

4.2.2 Unified Memory

To address these challenges, we evaluate Apple's Mac Studio M2 Ultra, as summarized in Table 4.3 and Figure 4.2. This system alleviates the aforementioned limitations and opens new research opportunities. The M2 Ultra is a highly cost-efficient consumer-grade machine, uniquely equipped with 128 GB (also available with 64GB and 192GB) of unified memory.

The term '**unified memory**' means that the system memory is shared between the CPU and the GPU, thereby eliminating the need for VRAM and explicit data transfers. In addition, its memory bandwidth of 819 GB/s significantly reduces memory bottlenecks. Notably, the combined performance of its CPU and integrated GPU is comparable to that of a mid-tier server CPU paired with a discrete GPU, all at a cost of just over USD 6,199.

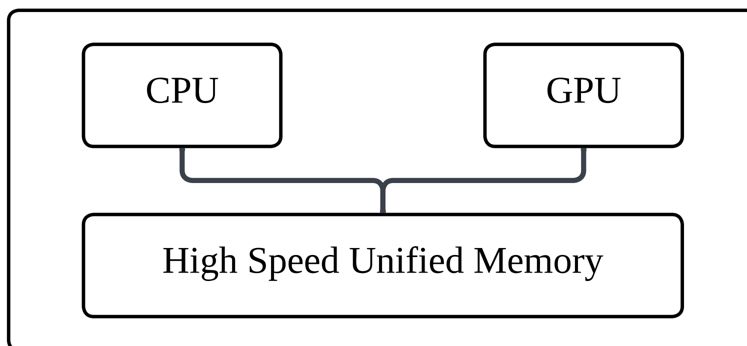

FIGURE 4.2: Illustration of Apple's Mac Studio M2 Ultra

CPU	Arm, 16 performance cores, 8 efficient cores
Last Level Cache	96 MB
GPU	76 cores, 9728 ALUs
Memory	128 GB LPDDR5, Unified, 819 GB/s
Storage	1TB
Price	USD 6,199

TABLE 4.3: Apple's Mac Studio M2 Ultra Specification

To illustrate the aforementioned data transfer cost and the advantages of unified memory, we conduct a simple integer sorting benchmark that can be easily configured and executed on a GPU. In this experiment, we vary the array size (the number of integers to be sorted) until it no longer fits into VRAM. Throughout this process, we observe the impact of data transfer both when VRAM is sufficient and when it is exhausted.

Why Sorting? Sorting is a fundamental operation that is widely used across applications. However, the type or nature of benchmark is less relevant. We only need a workload that actively operates on memory. Similar results can be obtained with other types of benchmarks.

	Discrete GPU	Integrated GPU
GPU	RTX 4070 SUPER	M2 Ultra
Channel	PCIe 4.0 x16	Unified memory
VRAM	12 GB dedicated	128 GB shared
Parallel Radix Sort	CUDA proprietary	Custom

TABLE 4.4: Test configuration of GPU integer sorting benchmark in figure 4.3
Note: RTX 4070 SUPER is selected because its compute power is close to, or slightly better than, M2 Ultra.

To ensure an apple-to-apple comparison, we select a discrete GPU that matches or slightly outperforms the M2 Ultra in compute capability specified in table 4.4, thereby minimizing the influence of computational differences and isolating the impact of memory and data movement. On both platforms, we employ parallel radix sort, however, the discrete GPU leverages an optimized proprietary library, while on the M2 Ultra we implemented a custom version due to the absence of library support.

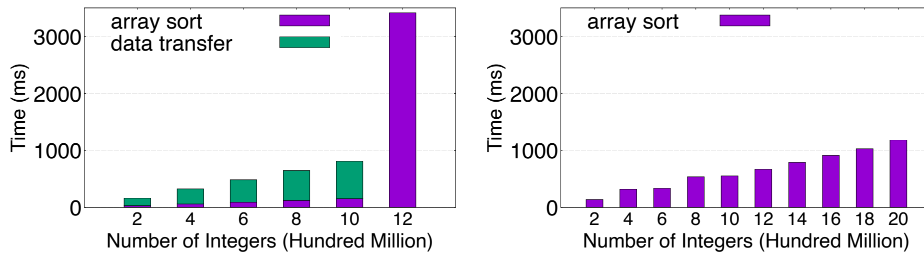

FIGURE 4.3: Time taken for GPU to sort an integer array. Lower the better. Left: Discrete GPU Right: Integrated GPU. Tests beyond 12 hundred million keys are omitted for discrete GPU as VRAM is already exhausted.

Figure 4.3 shows the time distribution breakdown for this benchmark. The array entirely fits within the discrete GPU's VRAM when keys number under one billion. Data movement across PCIe typically dominates execution time, accounting for more than 80% of the total. This pattern, where data are accessed once prior to being discarded, commonly exhibit in database operations. We also evaluate the circumstance when the array cannot fit VRAM. Sorting can still happen on discrete GPU within this scenario since the CUDA library transfers data transparently between host and GPU memory. The execution time increases by about

a factor of four, when array size is over one billion. In this case here, we are not able to determine the exact proportion of time that is spent moving data, as the library transfers data in a dynamic way whenever those data are required, rather than just initially and also finally transferring it simply.

For M2 Ultra in figure 4.3, the time required for the unified memory GPU to perform array sorting is greater than that of the discrete GPU. This result is expected due to possible differences in compute power and the potential for a less optimized sorting implementation. Nevertheless, by eliminating explicit data transfers, unified memory consistently outperforms the discrete GPU in all tested cases. Moreover, because the GPU has access to the entire memory space, it can efficiently sort significantly larger arrays without the limitation of VRAM capacity. And again, the M2 Ultra provides a unified memory capability that is uncommon in discrete GPU systems at a comparable cost.

4.2.3 Memory Speed

As shown in Table 4.3, the M2 Ultra's memory offers a bandwidth exceeding 800 GB/s. This bandwidth surpasses that of most server main memory and is comparable to the performance of high-end discrete GPU VRAM. To illustrate the extent of this speed, we present a detailed comparison between the M2 Ultra and a representative server configuration in this section.

It should be emphasized that the peak bandwidth of 800 GB/s is attainable only when multiple devices, such as the GPU, are utilized concurrently. When relying solely on the CPU, the achievable bandwidth is significantly lower and remains well below saturation, as will be further illustrated in Section 4.3. This section focuses on CPU-based evaluation to demonstrate the performance of the M2 Ultra's memory. Even when accessed by a single device, the system is capable of reaching memory speeds that are rarely observed in conventional architectures.

Memory performance typically falls within a number of key dimensions. The most influential among these are 1) sequential read 2) sequential write 3) memory copy and 4) latency [51, 149, 112]. In data-intensive workloads, sequential read and write bandwidth measure memory's efficiency when handling large, contiguous data transfers, a very critical factor. Memory copy

speed is equally significant, as it underlies many fundamental operations in database systems and is among the most frequently executed memory tasks. Latency, by contrast, reflects the responsiveness of memory to random accesses. This metric is particularly important for key-value stores, where pointer chasing and traversal across nodes dominate execution time.

CPU	Intel Xeon Gold 6230, 20 cores
Memory	64 GB DDR4 @ 2133 MT/s, 4 modules
Price	USD 4460

TABLE 4.5: Server Configuration

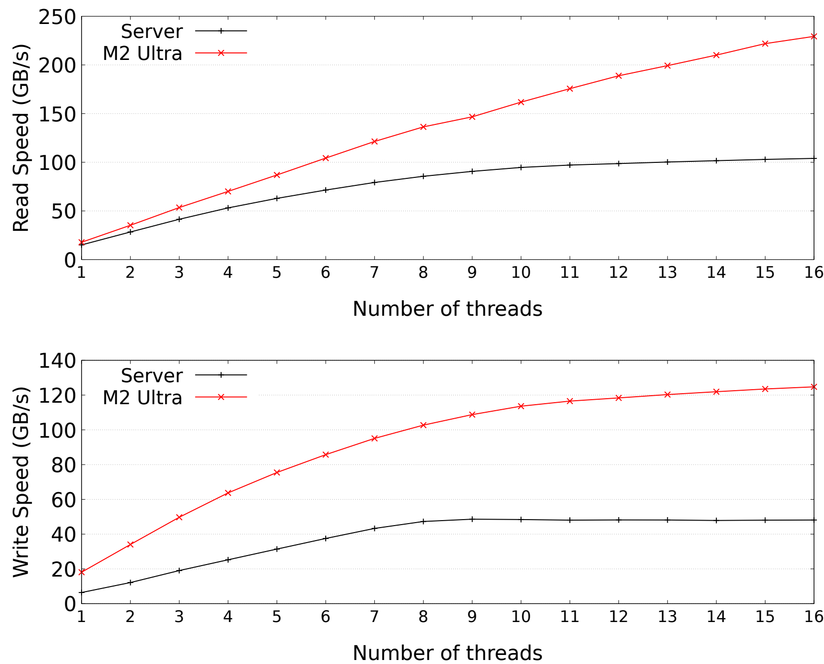

FIGURE 4.4: Sequential read(top) write(bottom) speed, higher the better.

Table 4.5 presents the configuration of the representative server used for comparison. This machine is equipped with DDR4 memory, a widely deployed class of main memory, operating across four channels on the motherboard.

Figures 4.4 offer a comparison of sequential write and read throughput between the two machines. Sequential read and write bandwidth, a key measure for memory subsystem

performance, refers to the rate at which contiguous memory blocks that are accessed or updated can be. Data-heavy applications value this specific metric. Many workloads do rely on high sequential bandwidth so as to move or to persist large volumes of data in an efficient way, and those workloads include table scans, log writing, checkpointing, data loading, and also large-scale analytical queries.

For the write benchmark, both devices throughput increases along as the number of threads grow, before eventually flattening out. The representative server's throughput in the read benchmark ceases to scale earlier with more threads. The M2 Ultra, however, continues to scale up with thread count. The M2 Ultra does achieve about double that of the representative server's throughput within practically every single configuration that was tested and also costs under double. A comparison that is fair is ensured by using CPUs on both of the platforms for the results. M2 Ultra's high-bandwidth memory is utilized more fully when GPU and CPU resources are in use together.

Indications: the M2 Ultra handles much greater sequential memory throughput than typical server platforms. Concurrency increases still when scaling is possible. Workloads with streaming large datasets such as machine learning pipelines, graph processing, in-memory analytics, log-structured merge (LSM) compactions, analytical aggregations, and bulk table scans find this capability especially helpful where high sequential read/write bandwidth translates directly into reduced execution times.

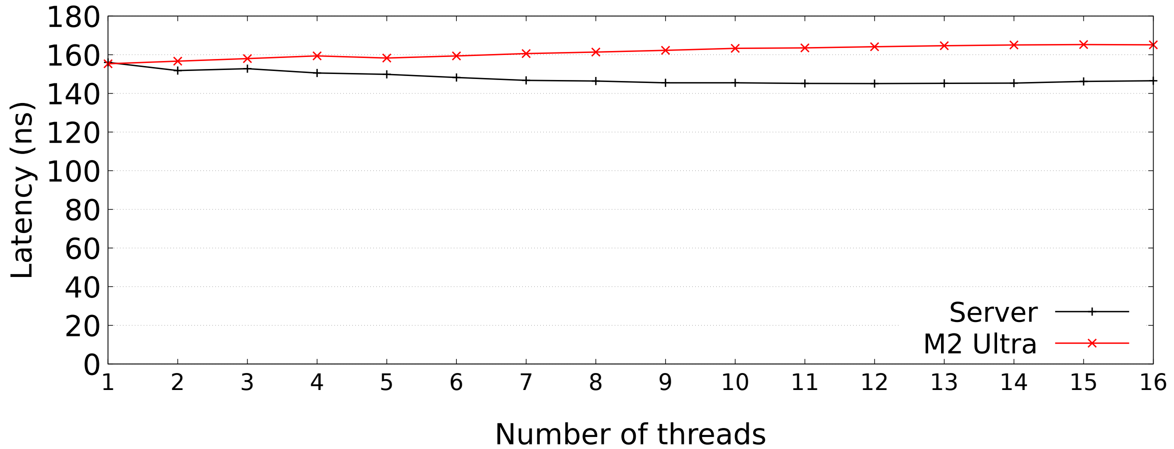

FIGURE 4.5: Latency for pointer chasing, lower the better.

Figure 4.5 presents a comparison between two machines in regard to pointer-chasing latency that was measured in nanoseconds. Pointer chasing refers to just a memory access pattern in which each memory access retrieves for itself the address of a next element, and then this creates a dependency chain of accesses. As this pattern widely evaluates random memory access latency, it prevents the processor from leveraging spatial locality or hardware prefetching. Commonly, it is employed for characterizing the latency of traversing linked lists and B-trees. It also characterizes such data structures as hash tables. Pointer-chasing latency can reflect the cost for dependent, fine-grained accesses. However, memory bandwidth measures the top sustained data transfer rate. Bandwidth as well as latency happen to be both properties regarding the memory subsystem however they are not correlated directly. High bandwidth does not necessarily imply low latency, though.

Both devices display about 150 ns of similar latencies per the results, with the M2 Ultra showing 20 ns more delay under higher thread counts. This result is noticeable since M2 Ultra provides much greater memory bandwidth than the discrete server machine however its latency traits are worse.

Indications: These observations suggest that customary pointer-chasing-operation-reducing approaches such as using larger nodes in B-tree structures to minimize traversal depth, remain effective on the M2 Ultra and comparable architectures. Thus, designers for such systems

should highlight designs using the plentiful memory bandwidth instead of depending on lower latency.

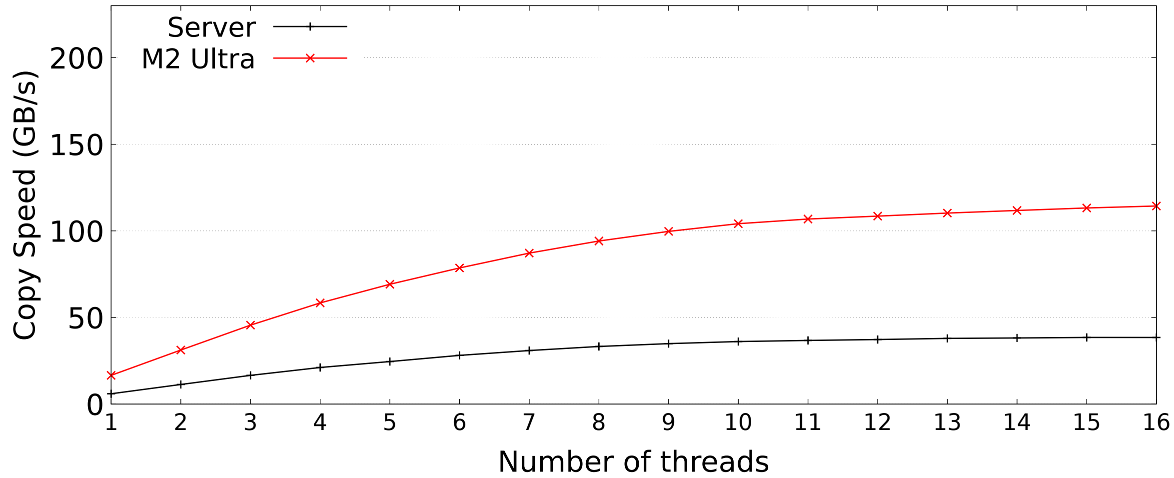

FIGURE 4.6: Speed of copying data, higher the better. Using library memcpy(), each of 1GB data.

Figure 4.6 presents a comparison concerning the throughput of memory copy between the two machines. Even though memory copying is conceptually simple enough, it is still one of the most fundamental operations and most frequent operations in database systems and in other data-intensive applications. It is underlying for a wide range of critical tasks. These tasks include moving intermediate query results, shuffling plus repartitioning data during joins, creating over merging buffers, also managing transaction logs alongside checkpointing. Memory copying speed directly affects total system performance since these operations often control execution time. Memory copy throughput has strong ties to effective bandwidth utilization because each copied byte must go through memory.

In this experiment the M2 Ultra reaches 114 GB/s while the representative server achieves about 38 GB/s peak memory copy throughput, 3× faster.

Indications: M2 Ultra is a very strong candidate for database workloads plus bandwidth-intensive apps. The massive copy speed is something that not only makes queries execute faster but also helps to load data, to construct indexes, to materialize query results, to analyze in-memory, and to handle large-scale scientific and machine learning workloads. Sort-merge

joins along with hash table construction also including matrix operations are workloads that do repeatedly move or transform large volumes of data. These workloads especially will obtain performance improvements that are important from such throughput of high memory copy.

4.3 Hash tables

We have examined the capabilities of unified memory and its high bandwidth in Section 4.2.3. Nevertheless, it remains difficult to determine how these characteristics translate to realistic workloads and, more critically, whether existing application designs can fully exploit this hardware. To address this, we begin by evaluating traditional index structure which has the potential to leverage these two properties, in order to identify which aspects of prior designs can be inherited, which should be discarded, and which require optimization for this emerging hardware.

Hashtable: The index structures based on GPU-resident hash tables are being leveraged in the state-of-the-art database systems. They combine fine-grained control over the placement of data and high levels of parallelism, making them popular with highly skilled developers. A GPU is a device with thousands of cores, and a hash table can take advantage of that by performing lookups simultaneously for many keys, achieving high throughput under good circumstances.

In this section, we extend **Mega-KV** [171], an existing fast hash table implementation for GPUs as illustrated in figure 4.7.

We briefly summaries Mega-KV as follows: Mega-KV adopts a bucketed cuckoo hashing, each bucket is fixed size, and each key can be placed in one of several candidate buckets by using multiple hash functions. Inside the buckets, instead of storing full keys or values, keys are compressed to a 32-bit signatures, and only the location ids of values are stored. On hash collisions, Mega-KV evicts existing entries to one of its alternate bucket positions, recursively

making room, similar to cuckoo eviction. Mega-KV is also optimized for memory alignment and parallelism on GPU by using aligned, fixed-size cells and buckets.

As a first step, we port Mega-KV to run on our new target, the GPU of Apple's M2 Ultra, and measure its behavior there (latency and throughput). We then make one step further by generalizing it, letting the CPU participate to boost throughput when pure GPU computation is no longer favorable.

We have encountered some performance oddities on the way. A few of these observations are counter-intuitive. We examine these anoma-lies closely and extract techniques which help to improve performance.

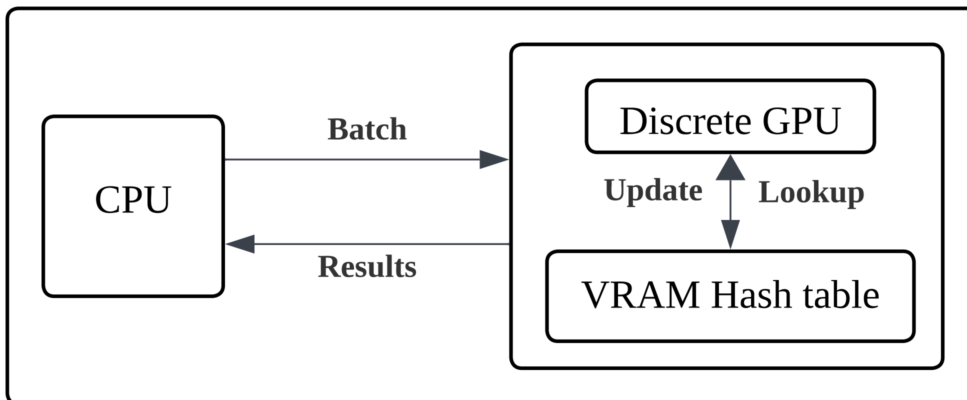

FIGURE 4.7: Design of Mega-KV, a GPU only hash table

Mostly, previous hash table implementations had used CPU or GPU not both concurrently. To leverage unified memory, the appearance of both devices (CPU and GPU) in a hash table is the first focus made by our work. The biggest challenge, and the most time consuming part, was porting the existing CUDA code over to Apple's Metal API, while also refitting x86/x64 architecture logic to run well on Arm (the architecture in Apple M-series chips). Although the porting of parallel kernels, memory fences, thread synchronization, sometimes rewriting of low-level device management is a heavy job and does not want to be underestimated in terms of time spent on the work. This thesis focuses far more on new findings and results obtained via all the approaches evaluated than actually on doing manual porting.

The hash table is essentially two operations: lookup and update (which is both insert and update). Lookups are simpler in dual-device setup as one can lookup from CPU and GPU simultaneously with such trivial coherence/consistency issues. But updates, particularly ones that change any shared buckets or rehash, are more complicated. Verifying that both devices update the table atomically is non-trivial (one has to resolve conflicts and maintain atomic, consistent and ordered updates), but one also want both the GPU write operations that made up a particular operation to be visible from CPU threads, and vice versa, when needed.

Listing 10 Pseudo-code of CPU verison of hashtable update

```
1  void cpu_update(key, value) {
2
3      while(!SUCCESS){
4          atomic_lock(bucket);
5
6          if (bucket[key] == EMPTY){
7              bucket[key] = value;
8              SUCCESS = true;
9          }else{
10             evict(bucket[key]);
11         }
12
13         atomic_unlock(bucket);
14     }
15 }
```

4.3.1 CPU-only

We first evaluate the performance of a naïve hash table originally ported from CUDA side of Mega-KV, running on the CPU of the M2 Ultra. In this version, we applied only the minimal modifications required for successful compilation and execution. Specifically, we employ fine-grained atomic locks, a commonly used approach, for updating buckets, as illustrated in Listing 10. We ensured that no changes were made to the overall design, such that the implementation reflects the true performance of the original design on different hardware. The atomic locks occupy a separate memory region, thus, updates to the hash table make exactly the same modifications as they would on a GPU, thereby enabling cooperation between CPU and GPU, which we will discuss in later sections.

Establishing this baseline is essential before assessing how much further performance can be improved and what trade-offs arise from porting complexity.

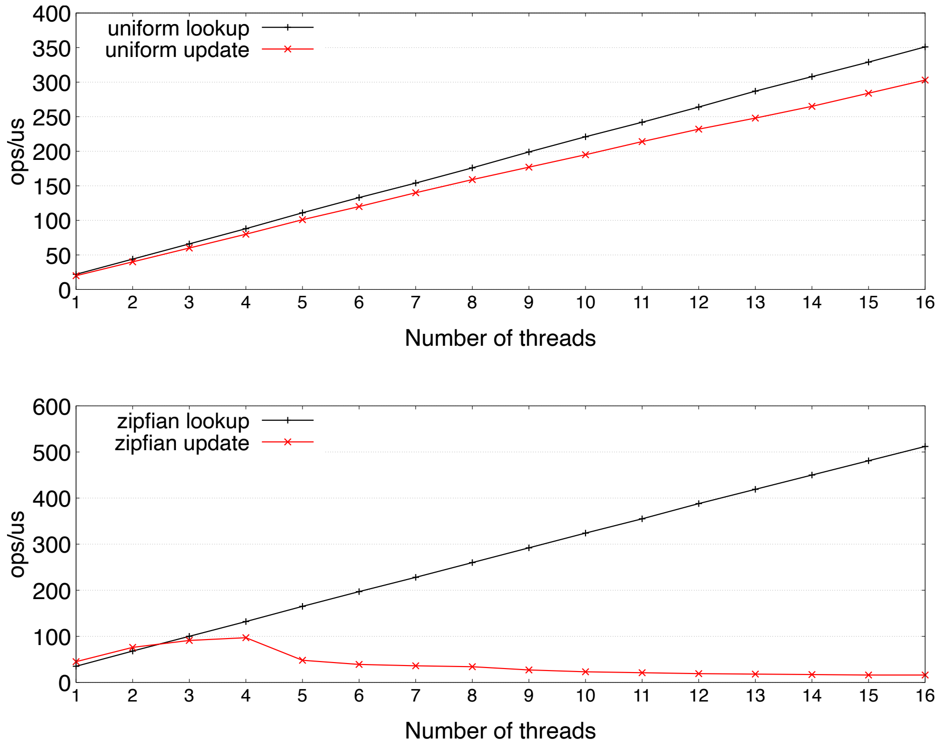

FIGURE 4.8: CPU hashtable performance, ported from Mega-KV. Using 16 threads, hashtable with 100 million keys. Uniform means random distribution of keys, where zipfian means skewed distribution of keys.

Figure 4.8 shows the performance of hash table ported for CPU. In one of our setups, with all 16 performance cores of the Apple M2 Ultra (not counting efficiency cores for now), CPU-only hash table yields surprisingly good results: about 350 operations per microsecond (ops/µs) on lookups and 303 ops/µs on updates. These numbers are based only on the ported code, without other optimization. We can compare against this baseline quite easily when exploring other hybrid or GPU powered architectures.

The most interesting result from the baseline is scalability. With a single CPU thread, we observe 22 ops/µs for lookup. With 16 CPU threads, we achieve nearly exactly 16× the speed (i.e., 352 ops/µs), indicating near-linear scaling. This is somewhat unusual on traditional systems. Typically when adding a lot of threads in an attempt to make the performance

character curve up, system is likely to get limited by memory bandwidth or synchronization overhead, causing the curve to trail down. Throughput just tends to level, or even degrade. But in this particular example the linear increase is an indication that the memory subsystem is very far from being saturated. It also shows that using yet one of the devices even if with many threads is underutilizing both the potential bandwidth and compute resources inherent to this platform.

When updating the hash table under a Zipfian key distribution, we observe that scalability breaks down due to contention on bucket locks. This limitation is expected, especially given that our CPU hash table implementation uses atomic locks in a straightforward manner without specialized optimizations. We chose not to optimize this case further, because workloads consisting of pure updates under a Zipfian distribution are uncommon in realistic application scenarios.

Indications: Our port of Mega-KV onto this unified/arm architecture gives competitive raw CPU performance. But using only CPU is clearly not insufficient to make the most from the powerful memory of M2 Ultra. One should aim for collaborating multiple devices.

4.3.2 GPU-only

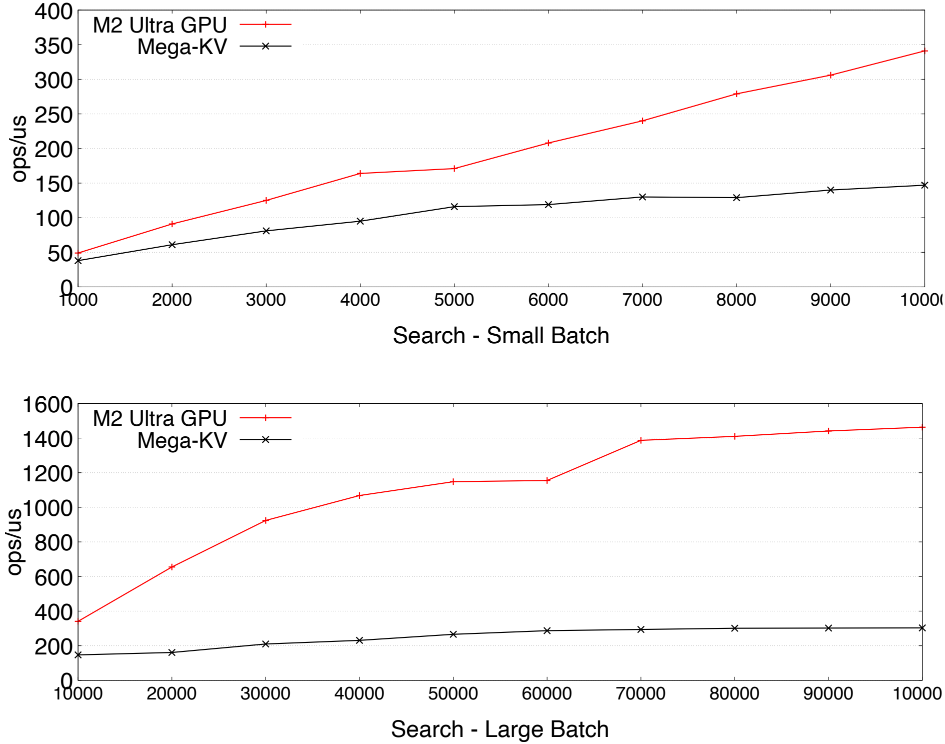

FIGURE 4.9: GPU hashtable performance, ported from Mega-KV. Varying batch size, hashtable with 100 million keys. Note: Performance of Mega-KV is copied from respective paper directly, due to unsuccessful launch using newer machines.

Figure 4.9 shows the performance of Mega-KV hash table ported from CUDA to Metal, for GPU. We evaluate our system under precisely the same variables that were used in the experiments for Mega-KV and for other state-of-the-art hash table systems. In particular, we measure how performance varies as we change batch size, since batch size often controls trade-offs between throughput, latency, and overhead in parallel hash table operations. We focus on lookup versus update throughput, kernel launch overheads, and how the integrated GPU of the Apple M2 Ultra compares with discrete GPUs in Mega-KV.

The M2 Ultra provides remarkably strong performance for an integrated GPU. Mega-KV paper uses two discrete GPUs, which are powerful but incur more overhead in terms of data transfer. Our results show that achieving high utilization on the embedded, memory shared

M2 Ultra GPU also requires selecting an appropriate batch size, similar to discrete GPUs. Small batches fail to saturate compute units or memory pipelines, excessively large batches can suffer increasing latency, queuing overhead, or contention. Thus, we confirm that batching is yet a good strategy to be inherited from traditional design.

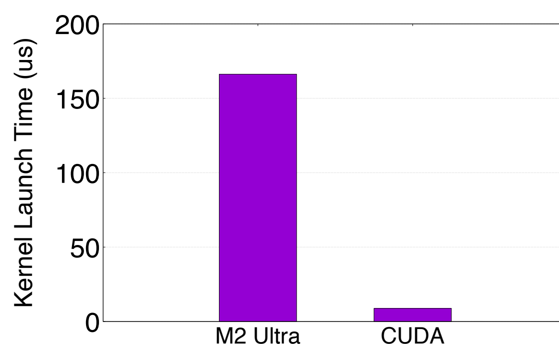

FIGURE 4.10: Compare of time to launch GPU kernel between M2 Ultra and CUDA. Lower the better.

Another observation that is something that further stresses the importance of batching is in relation to when the kernel launches, defining the time between dispatching a kernel and the time when it executes. Kernel launch time, as shown in Figure 4.10, differs greatly within the M2 Ultra's integrated GPU versus discrete GPUs. About 166 µs is the launch time on the M2 Ultra, but it is only around 9 µs on CUDA-enabled discrete GPUs. Frequently launching GPU kernels quickly is not efficient. So amassing and also sending larger batches of work is better therefore.

4.3.3 Combined CPU and GPU

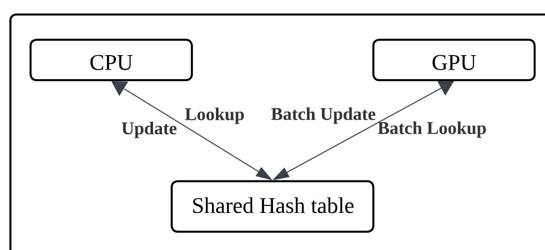

FIGURE 4.11: Design of extended hash table

In this section, we extend the hashtable from Mega-KV to utilize the unified memory architecture of M2 Ultra, by using multiple hardware as illustrated in figure 4.11.

4.3.3.1 Approach 1: Handing threads to the GPU

The first straightforward approach we employ is as follows: among the sixteen worker threads shown in Figure 4.8, we assign a subset of those threads to offload work to the GPU. Prior to examining more complex configurations or optimizations, we present an alternate perspective on the hash table's throughput under conditions of large batch size.

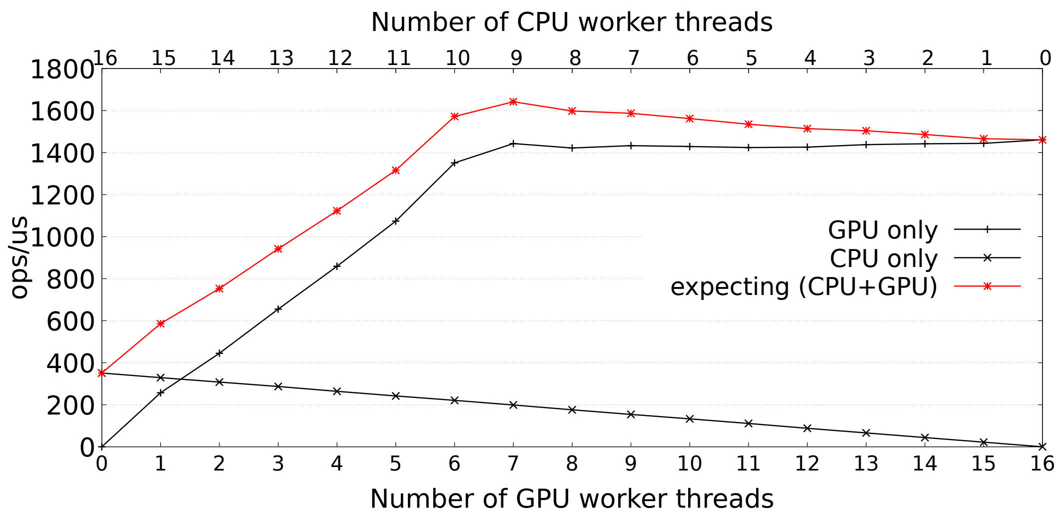

FIGURE 4.12: Hashtable throughput, using a total of 16 worker threads, varying number of GPU worker threads. When it increases, the number of CPU worker threads automatically goes down. Batch size for GPU worker threads is always large (100000 keys). 'Expecting' is thus the sum of two.

The performance depicted in figure 4.12 complements the CPU's hash table achieved in figure 4.8, together with GPU's using a consistent large batch size, allowing us to expect performance for both mixed GPU and CPU worker thread approaches under various combinations of worker threads. With this in mind, a noticeable performance improvement should be obtained by fully utilizing both hardware.

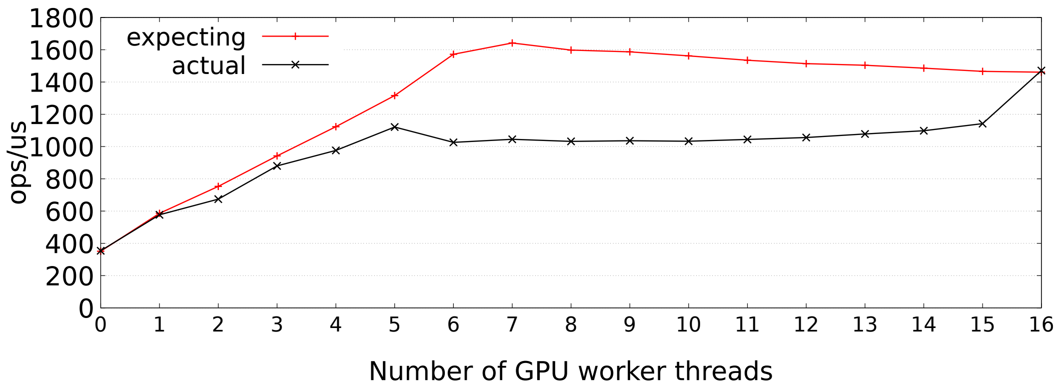

FIGURE 4.13: The actual bench marked throughput, derived from figure 4.12. The observed throughput is consistently lower than expected throughput.

In figure 4.13, the actual experimental results are presented. Unexpectedly, this combined approach does not give what is expected when both devices are used together. Underlying cause remains to be determined.

4.3.3.2 Approach 2: Using GPU idle time

In Approach 1, the GPU worker thread busy-waits while the GPU kernel is executing. We observed that this idle period constitutes a significant fraction of the execution time of a key-value store.

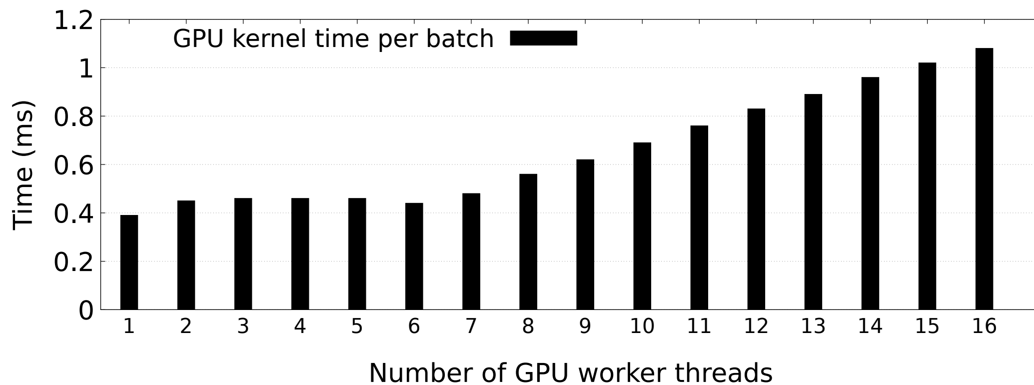

FIGURE 4.14: Per thread time spent on running a batch of 100000 lookup. Derived from curve of GPU from figure 4.12

As illustrated in Figure 4.14, the average CPU idle time while waiting for a large GPU batch ranges from 0.4 ms to over 1 ms, depending on the number of GPU threads competing for GPU resources. On traditional systems, it would not be feasible for the CPU to perform any useful work during this period. However, with unified memory, this becomes possible. We demonstrate a case in which the CPU performs its own hash table lookups while waiting for the GPU kernel to complete. In this approach, the CPU checks the status of the GPU kernel through a lightweight non-blocking function call, rather than blocking. It is important to note that not all of the GPU worker thread's idle time can be overlapped with CPU work. Only the period during which the GPU kernel is actively running can be overlapped, whereas argument preparation and kernel submission still occupy time that cannot be overlapped.

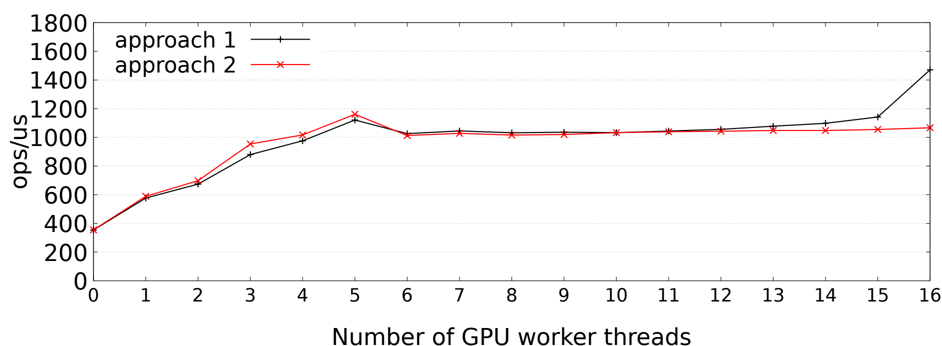

FIGURE 4.15: Hash table throughput comparison between approach 1 and 2. Higher the better.

Figure 4.15 compares the observed actual throughput of Approach 1 with that of Approach 2. Surprisingly, there is little to no improvement after applying Approach 2, in which the CPU utilizes the idle time while waiting for the GPU kernel. An important observation is that, when the number of GPU worker threads equals sixteen, all threads in Approach 1 are dedicated to GPU work, and no lookups occur on the CPU side. Nevertheless, Approach 1 achieves higher throughput, or, equivalently, does not incur any performance loss compared to single-device performance, as indicated by the 'GPU only' line in Figure 4.12. In contrast, when the CPU performs cooperative lookups in Approach 2, total throughput decreases, stabilizing at approximately one billion lookups per second. This suggests that the cooperation of the two devices, under the current design, results in suboptimal performance.

4.3.3.3 Issues with approach 1 and 2

Profiling reveals a substantial increase in the proportion of time devoted to reading from the hash table. No available performance counters provide an explanation for this behavior on the Apple M2 Ultra, and the native profiling tool, Apple's Instruments, similarly fails to identify the underlying cause.

Nevertheless, we were able to identify a general direction for addressing the problem and implement a practical solution based on the profiling results available. The issue appears to stem from the use of unified memory. A similar phenomenon observed on traditional CPUs is known as "false sharing," where multiple cores modify the same cache line concurrently. This situation is counterintuitive in our benchmark, as the hash table operations generate primarily read-only traffic, suggesting that cache line contention should not occur.

4.3.3.4 Approach 3: Replicating the hash table

We investigate this issue by assigning the CPU and GPU to access distinct memory regions. Previously, both devices operated on a fully shared hash table, with no copying or alternative address mapping applied. We then create an identical copy of the hash table after key insertion. This copy resides in a separate memory region, while remaining within the device's unified memory.

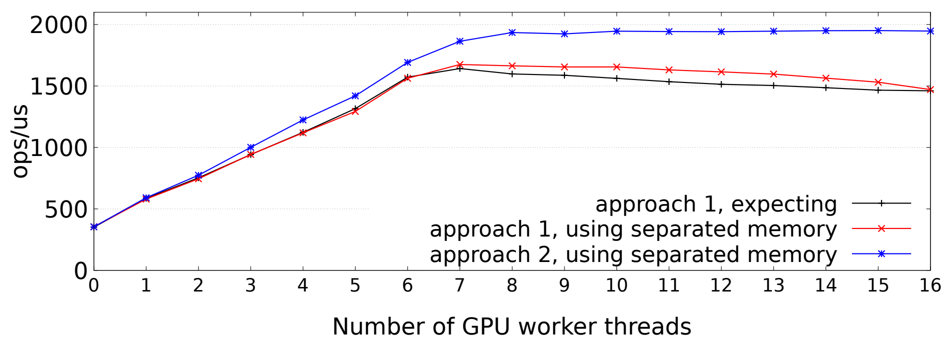

FIGURE 4.16: Hash table throughput of approach 1 and 2 when using separated memory region. Higher the better.

Surprisingly, as illustrated in Figure 4.16, both approaches significantly exceeded the previous and expected performance once distinct memory regions were employed. This outcome is anticipated for Approach 2, which leverages idle time for additional CPU lookups, but it is unexpected for Approach 1, as its predicted performance should represent an upper bound. Nevertheless, with this strategy implemented, the collaborative hash table achieves exceptional throughput, approaching nearly 2 billion lookups per second.

This results in a new design, illustrated in Figure 4.17 and listing 11. The rationale for this design is as follows: contemporary benchmarks, such as YCSB, indicate that workloads are predominantly read-heavy. Memory conflicts are observed even during lookups, but their impact is mitigated when an additional identical copy of the hash table is employed. Collisions during updates, however, cannot be entirely eliminated if both devices must observe the changes. Given that updates constitute only a small fraction of operations, this trade-off is considered acceptable, motivating the proposed design in which both hash tables are updated simultaneously.

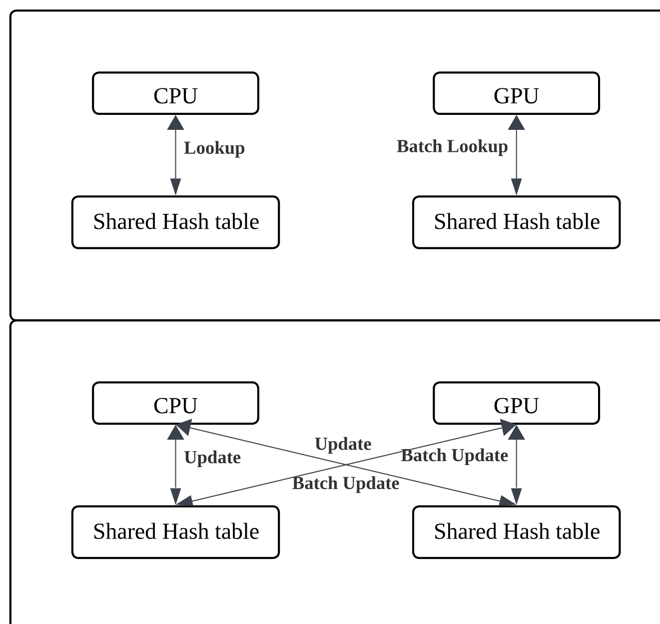

FIGURE 4.17: Design illustration of a duplicated hash table to improve performance with unified memory. When running lookups, each device accesses its own hash table. On the other hand, updates are replicated across both hash tables to ensure consistency.

Listing 11 Pseudo-code of simultaneous hash table update

```
1  void update(key, value) {
2    lock(bucket);
3    hashtable_CPU[key] = value;
4    hashtable_GPU[key] = value;
5    unlock(bucket);
6  }
```

Limitations: Using separate memory regions substantially improves performance but incurs a nearly unacceptable overhead in memory footprint, effectively doubling it. To address this, we propose a more efficient and practical approach, as illustrated in Figure 4.18.

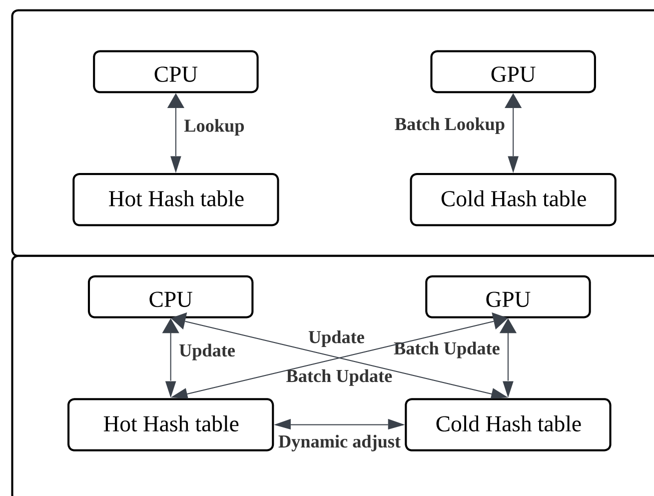

FIGURE 4.18: A smarter hashtable design. Instead of duplicating, CPU holds a smaller hash table with keys that frequently being accessed for optimal latency. GPU holds a larger hash table and complete big batches for high throughput. Memory footprint is therefore much lower.

Rather than duplicating the entire hash table, we move or replicate only a subset of the hottest keys or buckets, as most modern workloads exhibit skewed access patterns. These hot items are the primary source of memory conflicts. The CPU handles requests to these hot buckets to ensure minimal latency, while the GPU processes the remaining requests. Content is monitored and dynamically migrated between the two hash tables. Only when accesses are nearly uniform and confined to a relatively small hash table do we revert to the previous strategy of duplicating the entire structure.

As a preliminary step toward this solution, we validated this approach using a benchmark based on the design of a hot/cold hash table. The cache hierarchy of the M2 Ultra is as follows: the CPU own private L1 and L2 caches of 6.5 MB and 72 MB, and it share a large 96 MB L3 cache as the last-level cache (LLC) with GPU.

An intuitive design is for the CPU to operate a hash table that fits within the cache, while the GPU operates on the rest of the hash table. The regions owned by the two devices never overlap, so as to avoid the overhead of accessing the same memory region. The reason the CPU uses a small hash table is that it is commonly known that the CPU can only utilize a fraction of the total bandwidth, as illustrated in Figure 4.4 and 4.6. It also performs suboptimally due to memory stalls if prefetches are not issued properly, as will be explained later and shown in Figure 4.20. Hence, for the purpose of achieving the highest throughput while avoiding overheads, we propose this design.

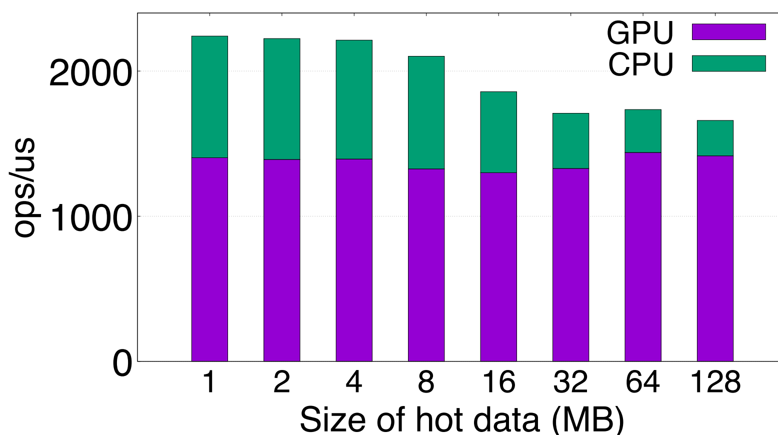

FIGURE 4.19: Benchmark test of changing the size of hot data which CPU will be working on. The size of data for GPU is unchanged.

Figure 4.19 presents the results of the benchmark. In this benchmark, we configured the key generator for lookups such that the CPU and GPU look up only a subset of keys. The keys are computed such that the buckets to which they belong span only a desired size of memory. Surprisingly, the throughput on the CPU side drops sharply only when it exceeds 4 MB, whereas we expected the hash table to remain in cache up to 96 MB. We suspect that this is due to the GPU concurrently accessing memory and thus polluting the cache, but we

have yet to verify this. Nevertheless, the high throughput of up to 2241 ops/µs provided by this solution validates the possibility of this approach while minimizing the memory footprint compared to a solution that duplicates the hash table.

4.3.4 Prefetching and its effects

Prefetching is a technique implemented at both the software and hardware level where it proactively loads data into the CPU cache before being requested. The data is already available in cache by the time that the CPU requires it, also this eliminates wait time thereby improving overall performance.

However, it is generally understood that prefetching can degrade overall performance in the event users misuse or overapply it. Prior studies [120, 101, 128, 177] show prefetching might hurt under heavy system load on usual platforms. In these cases, memcpy() or calls using lots of memory already utilize memory bandwidth heavily. However, our system does show that prefetching consistently yields gains without any degradation that we notice.

We use for evaluation the CPU-only hash table that Figure 4.8 depicts as a baseline, because prefetching is unavailable on the M2 Ultra's GPU. We vary the number of keys prefetched beforehand. The prefetch candidates correspond to the hash table buckets in order to retrieve associated values. Buckets distribute all through memory space, and the distribution makes a real scenario to evaluate prefetch.

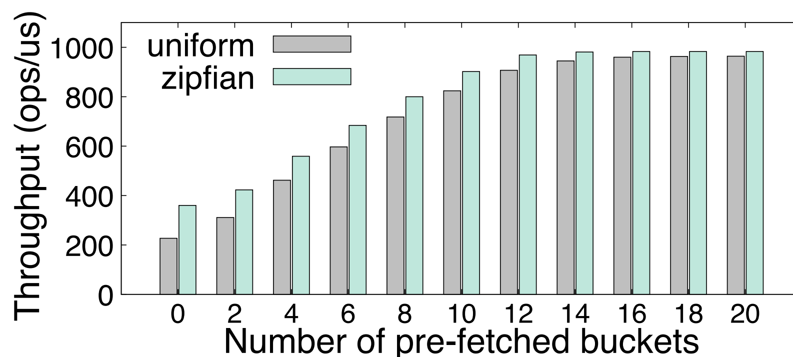

FIGURE 4.20: Hash table throughput with respect to number of prefetched buckets. Higher the better.

Benchmarking in figure 4.20 indicates that the application of prefetching can increase the baseline performance by nearly threefold when configured with a moderate prefetch distance of approximately 13 keys in advance, without any observable performance degradation. These results confirm that prefetching remains an effective optimization technique on this system. Consequently, we combine prefetching with the previously described strategies to maximize the overall performance of the hash table.

We also checked how sensitive and beneficial prefetching is for different access patterns. The results in Figure 4.20 show that even skewed accesses benefit significantly from prefetching, only with higher starting throughput.

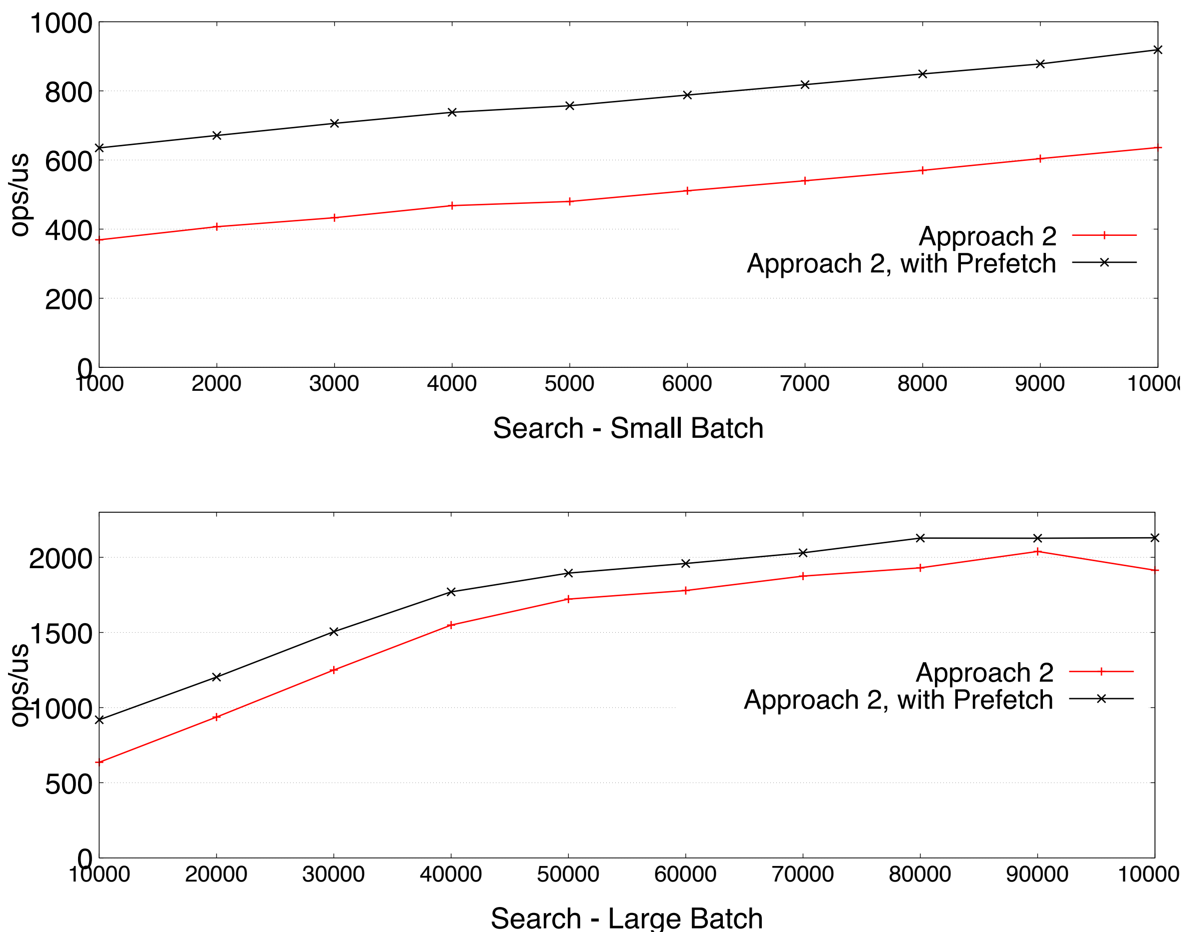

FIGURE 4.21: Performance of approach 2 with Prefetch added, 20 buckets are prefetched in advance.

There is evidence that hardware prefetchers exist on Apple Silicon. However, there is no publicly available way to control them. We can only add or remove software prefetches from the application. Prefetching does not degrade performance due to the large amount of available bandwidth, allowing prefetches to be performed without constraints.

We further went to integrate this technique into approach 2 with some separated memory regions that yielded the best results. Throughput with varying batch sizes is compared in Figure 4.21, like Figure 4.9. CPU prefetching offers more than a 70% throughput gain especially for small batch sizes with this setup. In these situations, CPU supplies most of the throughput.

We observed that there was no degradation when performance prefetched. Even in the case of larger batch sizes, where the GPU contributes to the majority of the throughput, prefetching still does provide measurable improvements. The relative performance gain decreases in percentage terms with batch size growth. Yet, the total rise in output stays almost uniform.

Summary: We ported a previously GPU-only hash table to run on both the CPU and GPU of the M2 Ultra and evaluated how effectively the two devices collaborate through unified memory. Based on profiling results, we proposed, implemented, and evaluated several designs to improve unified memory performance, including duplicated or dynamic hot/cold hash tables and the use of prefetching. Together, these solutions achieve multi-billion operations per second throughput.

4.3.5 Concurrently updating hashtable

Last but not least, we present our insights into concurrent updates of the hash table by both the CPU and GPU.

First, it is trivial that if we shard the hash table as aforementioned, updates will work without issue. However, this yields almost no significant contribution because, by simply separating all updates and lookups, the design falls back to that of a CPU and a discrete GPU, where

devices only access data over which they have ownership. We only remove the steps of copying data due to the nature of the hardware.

To make the most out of unified physical memory, we aim to achieve fine-grained updates, where the CPU and GPU can update any location of the hash table at any time. To achieve this, there are two possible approaches: (1) atomically compare and swap (hereafter referred to as CAS) words, and (2) protect the hash table buckets with locks.

Listing 12 Pseudo-code for testing cross-device atomics

```
1  void cpu_atomic(ptr) {
2
3      while(1){
4          atomic_fetch_add_explicit(ptr, memory_order_seq_cst, 1);
5          memory_barrier();
6      }
7  }
8
9  void gpu_atomic(ptr) {
10
11     while(1){
12
13         // only memory_order_weak is supported
14         atomic_fetch_add_explicit(ptr, memory_order_weak, 1);
15
16         atomic_thread_fence(device);
17         memory_barrier(device);
18     }
19 }
```

We first discuss approach 1. To accomplish this, it is necessary that one update to the hash table be achievable in a single step, namely one CAS, because two CAS operations can result in inconsistent results, as the first CAS does not guarantee claimed ownership. To verify this approach, we investigated the interaction of atomic operations on the M2 Ultra.

The CPU of the M2 Ultra supports the full range of atomic operations. Surprisingly, atomic operations on the GPU of M2 Ultra support only one memory ordering, namely weak memory ordering. To achieve full ordering with respect to other GPU threads, an explicit atomic thread fence must be used. We tested atomic operations on both devices separately, and they indeed worked within each device. However, problems arose when we attempted to coordinate

the two devices using atomic operations. Listing 12 demonstrates how we implemented cross-device atomics. Apple does not officially document system-level atomic operations, so instead we used the strongest available memory ordering for atomic operations on both sides. Furthermore, the GPU of the M2 Ultra has a configurable parameter to declare the cache-coherence scope of a piece of memory, which can be a SIMD group, thread group, or device level. The device level is the strongest among them and is the one we used. Even so, atomic operations failed to function properly when the two devices collaborated on the same variables. Therefore, we conclude for now that atomic operations cannot be used to synchronize the two devices and only work within a single device.

For approach 2, there are many implementations of locks, but there is no officially supported cross-device lock. Among programmer-implemented locks, some are atomic-based, but atomics do not work. Another option is to use memory barriers.

Listing 13 Pseudo-code for testing memory barriers

```
1  void threadA(A,B) {
2
3      write(A);
4      memory_barrier();
5      write(B);
6
7  }
8
9  void threadB(A,B) {
10
11     read(B);
12     memory_barrier();
13     read(A);
14 }
```

Memory barriers are a synchronization method used to ensure that updates are observed in order with respect to other cores on a device. Listing 13 shows a simple example of the use of memory barriers. Thread A updates A and then B. Without memory barriers, the compiler or hardware may reorder the updates of A and B. On another thread, it reads B and then A. If no reordering occurs, the thread should observe the update to B before the update to A. Memory barriers enforce that the updates issued by the first thread are observed in the correct

order from the perspective of B. If such ordering can be established, one can implement a lock using Lamport's bakery algorithm [150].

We therefore tested whether memory barriers could work across devices between the CPU and GPU. For both devices, we used the strongest available memory barriers. On the CPU, we used __sync_synchronize() and the dmb barrier. On the GPU, we used a thread-group barrier as well as an atomic thread fence. We instructed the functions to use the largest scope of coherence and the strongest memory ordering.

Surprisingly, the memory ordering only works in one direction. If the CPU performs updates and the GPU reads them, memory ordering is established. However, when reversed, the CPU cannot observe updates from the GPU in order, even with the aforementioned strong memory barriers. At the time this thesis is written, we are still deciding how to proceed with this finding.

Nevertheless, we aim to build a hash table that supports concurrent updates from both devices. An existing machine from AMD supports system-level atomics, and therefore we believe the contribution would be far less attractive if it were based on sharding.

4.4 Analytics Query

With the implementation and evaluation for key-value stores, the M2 Ultra, delivering strong performance in key-value store processing, becomes obvious with its unified memory architecture and high memory bandwidth.

Database systems fundamentally include key-value stores though. The performance by itself does not provide for a complete picture with respect to database behavior overall. It is simply because the operations that are in key-value stores are quite simple: retrieve the value(s) corresponding to any key by way of memory accesses, and potentially also read all of those values.

Listing 14 Pseudo-code of CPU verison of hashtable update

```
1   select l_returnflag ,
2          l_linestatus ,
3          sum( l_quantity ) as sum_qty , [...]
4   from lineitem
5   where l_shipdate <= "1998-09-04"
6   group by l_returnflag , l_linestatus
7   order by l_returnflag , l_linestatus ;
```

Databases do a wider range of operations instead. Listing 14 shows a simple, yet typical analytics SQL query within the TPC-H benchmark. This query needs operations that are more diverse which sort filter aggregate using hashing and retrieve values unlike key-value stores. For these operations, iterating over large datasets is required. Lightweight computations can also be performed in terms of these operations.

The M2 Ultra has a heterogeneous architecture that is well suited to characteristics such as those. For accelerating analytics queries an effective candidate is the platform since the CPU along with the GPU can process large data volumes because of unified memory plus high memory bandwidth without costly transfers.

Hence, to evaluate the analytical query performance of the M2 Ultra, we design, implement, and assess the query shown in Listing 14. We then compare our approach with the state-of-the-art database HeavyDB [61] to examine the extent to which unified memory can accelerate database queries.

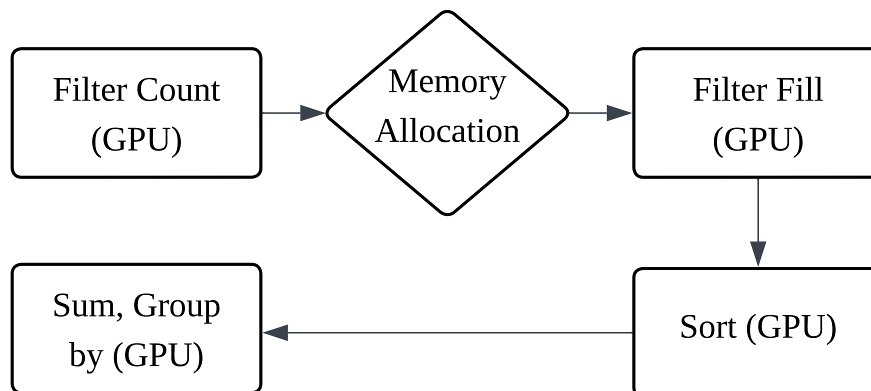

FIGURE 4.22: The design of running an analytics query on M2 Ultra.

Figure 4.22 depicts execution of Listing 14 in a custom workflow for M2 Ultra to fulfill the query. All operations are executed upon the GPU to exploit high parallelism as well as memory bandwidth, but critical tasks like result aggregation with memory allocation remain upon the CPU.

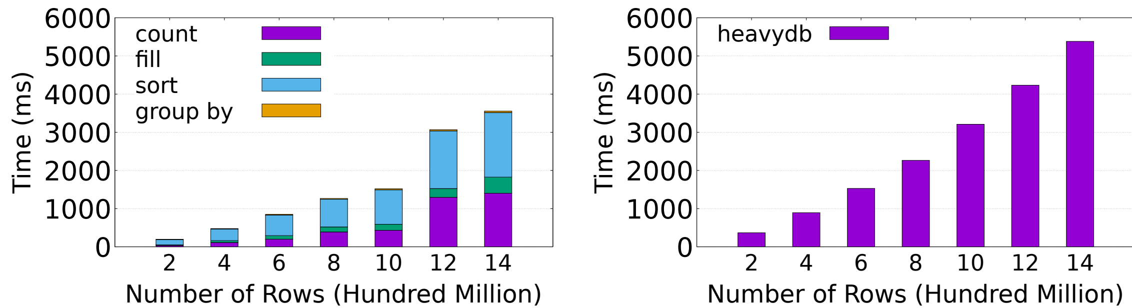

FIGURE 4.23: Time breakdown of running listing 14. Lower the better. Configurations are same as table 4.4.

As shown in Figure 4.23, even a straightforward design and implementation, potentially containing suboptimal code, enables the M2 Ultra to outperform its discrete GPU counterpart by reducing data movement. Furthermore, when the VRAM of a discrete GPU is exhausted (e.g., beyond 2 billion rows), HeavyDB falls back to CPU-based query processing, as discussed in Section 4.2.3, without leveraging managed memory. Consequently, further results are less comparable and are therefore omitted.

During implementation and evaluation, we identified an uncommon limitation of Metal, Apple's GPU programming framework: the absence of grid-level synchronization. Here, grid-level refers to synchronization across all GPU threads within a single kernel launch. In the absence of such a mechanism, synchronization can only be achieved by terminating the GPU kernel and performing aggregation on the CPU. This limitation increases the execution time of operations that inherently require global synchronization. For instance, result aggregation in the "Filter Count" operation must be performed on the CPU, and during parallel radix sort, the histogram of digit distributions between iterations must also be constructed on the CPU. As a result, execution time is less competitive compared to implementations using CUDA, which natively supports grid-level synchronization.

Nevertheless, as discussed in Section 4.2.3, it remains advantageous to employ the M2 Ultra's GPU. The elimination of explicit data transfers and the availability of high memory bandwidth offset these limitations, making the GPU execution model worthwhile despite Metal's constraints.

Implications: The high-speed unified memory of the M2 Ultra demonstrates strong potential even for database queries. However, substantial work remains. Kernel and launch code require further optimization, and the research community currently lacks common implementations and discussions regarding database query acceleration on such architectures.

4.5 Related Work

Integrated GPU: Several prior works [64, 63, 170, 38, 2] have accelerated key-value stores together with related data-intensive services through exploiting or evaluating integrated GPUs (iGPUs). Early efforts with memcached-style services ported to GPUs evaluated CPU+GPU deployments. These efforts highlighted both throughput gains and also challenges adapting server software to GPU execution models like MemcachedGPU and GNoM. These studies stress that using iGPU can reduce explicit data-copy overheads relative to discrete GPU setups, as they also repeatedly observe that the memory subsystem's behavior, shared system memory bandwidth, contention with CPU traffic, beside limited iGPU compute resources, can limit gains for certain workloads. The recent rediscoveries regarding iGPU platforms namely Apple's M-series SoCs renewed interest to measure how unified on-chip memory affects application performance as well as whether iGPUs are able to match mid-tier discrete GPUs for real workloads.

Discrete GPU: Since GPUs offer great compute throughput and bandwidth, many efforts [171, 77, 62, 95, 125] have used discrete GPUs to quicken key-value stores plus database kernels (Mega-KV represents large aggregate operation rates on CPU+GPU clusters). These systems use batched offload, pipeline designs, and careful data layout to maximize device utilization and hide latency. However, many of the studies and systems papers do also document such a recurring practical limit: host-device data moves across the PCIe (or other

links) and often becomes the dominant bottleneck when the working sets exceed VRAM or when such offloads are fine-grained. Papers examine GPU query processing as well as large-data processing on GPUs. These papers specifically cite PCIe as the limit, so they foster designs avoiding frequent transfers via locality, batching, or NVLink or rearchitect systems lessening host, device communication.

Unified, high-bandwidth memory: Recent hardware trends like Apple unifies memory in M-series SoCs and CXL and other coherent links emerge motivate rethinking memory/system design in data systems. Database buffer management with query processing plus scale-up engine designs happen to be affected through coherent high-capacity remote memory just as investigated through work upon CXL along with disaggregated/shared memory. Recent papers [144, 165, 176, 149, 164, 92] fully explore all the opportunities plus performance implications from CXL-backed main memory. Apart from this, experimental evaluations [67] regarding Apple's unified memory (and early M-series benchmarking) have started to quantify just how removing explicit host-device copies along with having a very high aggregate bandwidth can change performance tradeoffs within memory-bound kernels. These lines of work are quite active and quite growing, yet still relatively few thorough studies specifically evaluate key-value stores or else hash-based indices on UMA (Uniform Memory Access) platforms. So the literature motivates questions we address regarding practical KV-store design on unified high-bandwidth machines but does not fully cover these.

4.6 Conclusion

We have studied a system featuring an evolving memory architecture. To explore its unique characteristics of unified and high-speed memory, we ported and evaluated a previously GPU-only hash table. We further optimized it with several design adaptations and techniques to ensure it operates effectively on this memory type. Our findings also suggest that such architectures hold significant untapped potential for database systems, which remains to be fully explored.

CHAPTER 5

Final Remarks and Future Work

In this thesis, we have provided an intensive review of PMem, showing why traditional approaches might be outdated and unfit for evolving memory. With Pre-stores and DirtBuster, we help programmers identify and counter performance anomalies in their applications, making the most out of heterogeneous memory. The study of high-speed unified memory demonstrates that it has wide potential applications. However, its use is not straightforward, it has to be done carefully. At present, academia has only a shallow understanding of it.

In the future, I aim to complete the preliminary studies of high-speed unified memory, namely to complete Chapter 4 and submit it to conferences. I believe there is much more work to be done and many more papers to be published, as this type of memory will become the norm with the dramatic demand for memory across multiple devices. There will also be more work to be done with Pre-stores. I hope that we can further extend it to be a compiler built-in or enabled in hardware as hardware Pre-stores. A more near-term plan is to experiment with Pre-stores on even more memory systems.

Thank you for reading my thesis, and I hope you find it insightful.

Bibliography

[1] Inc Advanced Micro Devices. *AI Acceleration With AMD Radeon*. https://www.amd.com/en/products/graphics/radeon-ai.html. 2024.

[2] Homa Aghilinasab et al. 'Dynamic memory bandwidth allocation for real-time GPU-based SoC platforms'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3348–3360.

[3] Ghadeer Almusaddar and Hoda Naghibijouybari. 'Exploiting parallel memory write requests for covert channel attacks in integrated cpu-gpu systems'. In: *arXiv preprint arXiv:2307.16123* (2023).

[4] Eric Anderson et al. 'What Consistency Does Your {Key-Value} Store Actually Provide?' In: *Sixth Workshop on Hot Topics in System Dependability (HotDep 10)*. 2010.

[5] Mufakir Qamar Ansari and Mudabir Qamar Ansari. 'Racing to Idle: Energy Efficiency of Matrix Multiplication on Heterogeneous CPU and GPU Architectures'. In: *arXiv preprint arXiv:2507.20063* (2025).

[6] Arm. *L210 Cache Controller Technical Reference Manual*. https://developer.arm.com/documentation/ddi0284/g/Babeegje. 2023.

[7] Joy Arulraj et al. 'BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory'. In: *Proceedings of the VLDB Endowment* 11.5 (2018).

[8] Berk Atikoglu et al. 'Workload analysis of a large-scale key-value store'. In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 2012, pp. 53–64.

[9] Reza Azimi et al. 'Enhancing operating system support for multicore processors by using hardware performance monitoring'. In: *ACM SIGOPS Operating Systems Review* 43.2 (2009), pp. 56–65.

[10] David Bailey et al. *The NAS parallel benchmarks 2.0*. Tech. rep. Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[11] Oana Balmau et al. 'SILK+: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores Running Heterogeneous Workloads'. In: *ACM Transactions on Computer Systems (TOCS)* 36.4 (2020).

[12] Oana Balmau et al. 'TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores'. In: *Proceedings of USENIX ATC*. 2017.

[13] Lawrence Benson, Hendrik Makait and Tilmann Rabl. 'Viper: An efficient hybrid pmem-dram key-value store'. In: *Proceedings of the VLDB Endowment* 14.9 (2021), pp. 1544–1556.

[14] Mateusz Berezecki et al. 'Many-core key-value store'. In: *2011 International Green Computing Conference and Workshops*. IEEE. 2011, pp. 1–8.

[15] Brian N Bershad et al. 'Avoiding conflict misses dynamically in large direct-mapped caches'. In: *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*. 1994, pp. 158–170.

[16] Kumud Bhandari, Dhruva R Chakrabarti and Hans-J Boehm. 'Makalu: Fast recoverable allocation of non-volatile memory'. In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 677–694.

[17] Daniel Bittman et al. 'Twizzler: a Data-Centric OS for Non-Volatile Memory'. In: *Proceedings of USENIX ATC*. 2020.

[18] Edouard Bugnion et al. 'Compiler-directed page coloring for multiprocessors'. In: *ACM SIGPLAN Notices* 31.9 (1996), pp. 244–255.

[19] Wentao Cai et al. 'Understanding and Optimizing Persistent Memory Allocation'. In: *Proceedings of ISMM*. 2020.

[20] Jiashen Cao et al. 'Gpu database systems characterization and optimization'. In: *Proceedings of the VLDB Endowment* 17.3 (2023), pp. 441–454.

[21] Guilherme Cassales et al. 'Improving the performance of bagging ensembles for data streams through mini-batching'. In: *Information Sciences* 580 (2021), pp. 260–282.

[22] Roberto Cavicchioli, Nicola Capodieci and Marko Bertogna. 'Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms'. In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2017, pp. 1–10.

[23] Helen HW Chan et al. 'Hashkv: Enabling efficient updates in {KV} storage via hashing'. In: *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 2018, pp. 1007–1019.

[24] Hanqiu Chen et al. 'Bottleneck analysis of dynamic graph neural network inference on cpu and gpu'. In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2022, pp. 130–145.

[25] Shimin Chen and Qin Jin. 'Persistent B+-trees in Non-volatile Main Memory'. In: *Proceedings of the VLDB Endowment* 8.7 (2015).

[26] William Y Chen et al. 'An efficient architecture for loop based data preloading'. In: *ACM SIGMICRO Newsletter* 23.1-2 (1992), pp. 92–101.

[27] Youmin Chen et al. 'FlatStore: An efficient log-structured key-value storage engine for persistent memory'. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1077–1091.

[28] David Cock et al. 'Enzian: an open, general, CPU/FPGA platform for systems software research'. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 434–451.

[29] Many contributors. *Samsung Electronics Introduces Industry's First 512GB CXL Memory Module*. 'https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module'. 2022.

[30] Wikipedia Contributors. *Graphics Address Remapping Table*. Accessed: 2025-09-25. 2025. URL: https://en.wikipedia.org/wiki/Graphics_address_remapping_table.

[31] Intel Corporation. *A utility for configuring and managing Intel® Optane™ Persistent Memory modules (PMem)*. https://github.com/intel/ipmctl. 2024.

[32] Intel Corporation. *Discover Advanced Memory with Intel® Optane™ PMem*. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html. 2024.

[33] Intel Corporation. *From Intel® Optane™ Persistent Memory to CXL*. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory-to-cxl-attached-memory.html. 2024.

[34] Intel Corporation. *Intel® Accelerator Engines for Demanding Workloads Help Enhance ROI*. 2024.

[35] Intel Corporation. *Intel® Xeon® CPU Max Series*. https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html. 2024.

[36] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html. 2024.

[37] Zheng Dang et al. 'Nvalloc: Rethinking heap metadata management in persistent memory allocators'. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 115–127.

[38] Mohammad Dashti and Alexandra Fedorova. 'Analyzing memory management methods on integrated CPU-GPU systems'. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 2017, pp. 59–69.

[39] Tudor David, Rachid Guerraoui and Vasileios Trigonakis. 'Asynchronized concurrency: The secret to scaling concurrent search data structures'. In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pp. 631–644.

[40] Niv Dayan, Manos Athanassoulis and Stratos Idreos. 'Monkey: Optimal navigable key-value store'. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 79–94.

[41] Giuseppe DeCandia et al. 'Dynamo: Amazon's highly available key-value store'. In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[42] Anthony Demeri et al. 'Poseidon: Safe, fast and scalable persistent memory allocator'. In: *Proceedings of the 21st International Middleware Conference*. 2020, pp. 207–220.

[43] DigitalOcean. *GPU Memory Bandwidth and Its Impact on Performance*. Accessed: 2025-09-25. 2025. URL: https://www.digitalocean.com/community/tutorials/gpu-memory-bandwidth.

[44] Digitimes. 'PCIe Gen5 in high-end PCs set to drive Silicon Motion's sales growth in 2025'. In: *Digitimes* (2025). Accessed: 2025-09-25. URL: https://www.digitimes.com/news/a20250213PD205/silicon-motion-pcie-pc-growth-2025.html.

[45] Zhimin Ding et al. 'Turnip: A nondeterministic gpu runtime with cpu ram offload'. In: *arXiv preprint arXiv:2405.16283* (2024).

[46] Bestware Documentation. *Allocation of video memory in systems with integrated graphics units (iGPU)*. https://help.bestware.com/hc/en-gb/articles/30053124786333-Allocation-of-video-memory-in-systems-with-integrated-graphics-units-iGPU. 2024.

[47] The Linux Kernel Documentation. *Linux Kernel Module Parameters amdgpu vram-limit vis vramlimit*. https://docs.kernel.org/gpu/amdgpu/module-parameters.html. 2024.

[48] Zhuohui Duan et al. 'Revisiting Log-Structured Merging for KV Stores in Hybrid Memory Systems'. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 674–687.

[49] Easyshoppi. *Is PCI Express 3.0 Still Relevant in 2025?* Accessed: 2025-09-25. 2025. URL: https://easyshoppiblog.wordpress.com/2025/04/04/is-pci-express-3-0-still-relevant-in-2025/.

[50] Robert Escriva, Bernard Wong and Emin Gün Sirer. 'HyperDex: A distributed, searchable key-value store'. In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 2012, pp. 25–36.

[51] Pouya Esmaili-Dokht et al. 'A mess of memory system benchmarking, simulation and application profiling'. In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2024, pp. 136–152.

[52] Facebook. *RocksDB: An Embeddable Persistent Key-Value Store for Fast Storage*. https://https://rocksdb.org/. [Online; accessed 17-July-2020]. 2013.

[53] Jian Fang, Han Chen, Jian Mao et al. 'Understanding Data Partition for Applications on CPU-GPU Integrated Processors'. In: *Mobile Ad-hoc and Sensor Networks (MSN 2017)*. Vol. 747. Communications in Computer and Information Science. Springer, 2018, pp. 426–434. DOI: 10.1007/978-981-10-8890-2_32.

[54] Dahua Feng et al. 'Profiling Apple Silicon Performance for ML Training'. In: *arXiv preprint arXiv:2501.14925* (2025).

[55] Diogo Flores. *X9 - High performance message passing library*. https://github.com/df308/x9. 2023.

[56] João Gonçalves, Miguel Matos and Rodrigo Rodrigues. 'Mumak: Efficient and Black-Box Bug Detection for Persistent Memory'. In: *Proceedings of the Eighteenth European Conference on Computer Systems*. 2023, pp. 734–750.

[57] Alfonso Mascareñas González et al. 'Exploring iGPU Memory Interference Response to L2 Cache Locking'. In: *21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*. Vol. 114. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2023, pp. 3–1.

[58] Tobias Groth et al. 'Hybrid CPU/GPU/APU accelerated query, insert, update and erase operations in hash tables with string keys'. In: *Knowledge and Information Systems* 65.10 (2023), pp. 4359–4377.

[59] Gaël Guennebaud, Benoît Jacob et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[60] Khan Shaikhul Hadi et al. 'The Case for Persistent CXL switches'. In: *arXiv preprint arXiv:2503.04991* (2025).

[61] HeavyAI. *HeavyDB (GitHub Repository)*. https://github.com/heavyai/heavydb. Accessed: 2025-09-25. 2025.

[62] Steef Hegeman et al. 'Compact Parallel Hash Tables on the GPU'. In: *European Conference on Parallel Processing*. Springer. 2024, pp. 226–241.

[63] Tayler H Hetherington, Mike O'Connor and Tor M Aamodt. 'Memcachedgpu: Scaling-up scale-out key-value stores'. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2015, pp. 43–57.

[64] Tayler H Hetherington et al. 'Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems'. In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE. 2012, pp. 88–98.

[65] Qingda Hu et al. 'Log-Structured Non-Volatile Main Memory.' In: *USENIX Annual Technical Conference*. 2017, pp. 703–717.

[66] Yihe Huang et al. 'Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs'. In: *Proceedings of USENIX ATC*. 2018.

[67] Paul Hübner et al. 'Apple vs. Oranges: Evaluating the Apple Silicon M-Series SoCs for HPC Performance and Efficiency'. In: *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2025, pp. 45–54.

[68] Deukyeon Hwang et al. 'Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree'. In: *Proceedings of FAST*. 2018.

[69] Apple Inc. *Apple introduces M4 chip*. https://www.apple.com/au/newsroom/2024/05/apple-introduces-m4-chip/. 2024.

[70] Analyzing Memory Management Methods on Integrated CPU-GPU Systems. 'Analyzing Memory Management Methods on Integrated CPU-GPU Systems'. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM)*. 2017. DOI: 10.1145/3092255.3092256.

[71] Intel. *Persistent Memory Development Kit*. http://pmem.io/. [Online; accessed 12-November-2022]. 2020.

[72] Intel. *PMemKV: Intel's key-value datastore optimized for persistent memory*. https://github.com/pmem/pmemkv. [Online; accessed 17-July-2020]. 2017.

[73] Yasuo Ishii, Mary Inaba and Kei Hiraki. 'Access map pattern matching for data cache prefetch'. In: *Proceedings of the 23rd international conference on Supercomputing*. 2009, pp. 499–500.

[74] Joseph Izraelevitz et al. ‘Basic performance measurements of the intel optane DC persistent memory module’. In: *arXiv preprint arXiv:1903.05714* (2019).

[75] Reiley Jeyapaul and Aviral Shrivastava. ‘Smart cache cleaning: Energy efficient vulnerability reduction in embedded processors’. In: *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. 2011, pp. 105–114.

[76] Chaoyi Jiang et al. ‘KVPR: Efficient LLM Inference with I/O-Aware KV Cache Partial Recomputation’. In: *arXiv preprint arXiv:2411.17089* (2024).

[77] Daniel Jünger et al. ‘Warpcore: A library for fast hash tables on gpus’. In: *2020 IEEE 27th international conference on high performance computing, data, and analytics (HiPC)*. IEEE. 2020, pp. 11–20.

[78] Olzhas Kaiyrakhmet et al. ‘{SLM-DB}:{Single-Level}{Key-Value} store with persistent memory’. In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 2019, pp. 191–205.

[79] Woosung Kang et al. ‘RT-Swap: Addressing GPU Memory Bottlenecks for Real-Time Multi-DNN Inference’. In: *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society. 2024, pp. 373–385.

[80] Sudarsun Kannan et al. ‘Redesigning LSMs for Nonvolatile Memory with NoveLSM’. In: *Proceedings of USENIX ATC*. 2018.

[81] Richard E Kessler and Mark D Hill. ‘Page placement algorithms for large real-indexed caches’. In: *ACM Transactions on Computer Systems (TOCS)* 10.4 (1992), pp. 338–359.

[82] Samira Khan et al. ‘Improving cache performance using read-write partitioning’. In: *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*. IEEE. 2014, pp. 452–463.

[83] Shun Kida, Satoshi Imamura and Kenji Kono. ‘Revisiting Memory Swapping for Big-Memory Applications’. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 2025, pp. 33–42.

[84] Jin-Young Kim et al. 'An effective pre-store/pre-load method exploiting intra-request idle time of NAND flash-based storage devices'. In: *Microprocessors and Microsystems* 50 (2017), pp. 222–236.

[85] Wook-Hee Kim et al. 'PACTree: A high performance persistent range index using PAC guidelines'. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 424–439.

[86] Rakesh Krishnaiyer et al. 'Compiler-based data prefetching and streaming non-temporal store generation for the intel (r) xeon phi (tm) coprocessor'. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE. 2013, pp. 1575–1586.

[87] Miryeong Kwon et al. 'From Block to Byte: Transforming PCIe SSDs with CXL Memory Protocol and Instruction Annotation'. In: *IEEE Micro* (2025).

[88] Se Kwon Lee et al. 'Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes'. In: *Proceedings of SOSP*. 2019.

[89] Se Kwon Lee et al. 'WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems'. In: *Proceedings of FAST*. 2017.

[90] Baptiste Lepers and Willy Zwaenepoel. 'Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems)'. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, July 2023, pp. 519–534. ISBN: 978-1-939133-34-2. URL: https://www.usenix.org/conference/osdi23/presentation/lepers.

[91] Baptiste Lepers et al. 'KVell: The Design and Implementation of a Fast Persistent Key-Value Store'. In: *Proceedings of SOSP*. 2019.

[92] Alberto Lerner and Gustavo Alonso. 'Cxl and the return of scale-up database engines'. In: *arXiv preprint arXiv:2401.01150* (2024).

[93] Lucas Lersch et al. 'Enabling low tail latency on multicore key-value stores'. In: *Proceedings of the VLDB Endowment* 13.7 (2020), pp. 1091–1104.

[94] Justin J. Levandoski, David B. Lomet and Sudipta Sengupta. 'The Bw-Tree: A B-Tree for New Hardware Platforms'. In: *Proceedings of ICDE*. 2013.

[95] Yinan Li et al. ‘Scaling GPU-Accelerated Databases beyond GPU Memory Size’. In: *Proceedings of the VLDB Endowment* 18.11 (2025), pp. 4518–4531.

[96] Youjie Li et al. ‘Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers’. In: *arXiv preprint arXiv:2202.01306* (2022).

[97] Hyeontaek Lim et al. ‘MICA: A Holistic Approach to Fast In-Memory Key-Value Storage’. In: *Proceedings of NSDI*. 2014.

[98] Hyeontaek Lim et al. ‘SILT: A memory-efficient, high-performance key-value store’. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 1–13.

[99] Sihang Liu et al. ‘PMTest: A fast and flexible testing framework for persistent memory programs’. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 411–425.

[100] Xinjian Long et al. ‘An intelligent framework for oversubscription management in cpu-gpu unified memory’. In: *Journal of Grid Computing* 21.1 (2023), p. 11.

[101] Xinjian Long et al. ‘Deep learning based data prefetching in CPU-GPU unified virtual memory’. In: *Journal of Parallel and Distributed Computing* 174 (2023), pp. 19–31.

[102] Baotong Lu et al. ‘Dash: Scalable Hashing on Persistent Memory’. In: *Proceedings of the VLDB Endowment* 13.8 (2020).

[103] Chi-Keung Luk and Todd C Mowry. ‘Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors’. In: *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE. 1998, pp. 182–193.

[104] Francesco Lumpp, Hiren D Patel and Nicola Bombieri. ‘A framework for optimizing cpu-igpu communication on embedded platforms’. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 685–690.

[105] Shaonan Ma et al. ‘ROART: Range-query Optimized Persistent ART.’ In: *FAST*. 2021, pp. 1–16.

[106] Yandong Mao, Eddie Kohler and Robert Tappan Morris. 'Cache craftiness for fast multicore key-value storage'. In: *Proceedings of the 7th ACM european conference on Computer Systems*. 2012, pp. 183–196.

[107] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[108] Phoronix Media. *Open-Source, Automated Benchmarking*. https://www.phoronix-test-suite.com/. 2023.

[109] Alexander Merritt et al. 'Concurrent log-structured memory for many-core key-value stores'. In: *Proceedings of the VLDB Endowment* 11.4 (2017), pp. 458–471.

[110] Sparsh Mittal. 'A survey of recent prefetching techniques for processor caches'. In: *ACM Computing Surveys (CSUR)* 49.2 (2016), pp. 1–35.

[111] Todd C Mowry. 'Tolerating latency in multiprocessors through compiler-inserted prefetching'. In: *ACM Transactions on Computer Systems (TOCS)* 16.1 (1998), pp. 55–92.

[112] Onur Mutlu and Lavanya Subramanian. 'Research problems and opportunities in memory systems'. In: *Supercomputing frontiers and innovations* 1.3 (2014), pp. 19–55.

[113] Moohyeon Nam et al. 'Write-optimized Dynamic Hashing for Persistent Memory'. In: *Proceedings of FAST*. 2019.

[114] Faisal Nawab et al. 'Dalí: A Periodically Persistent Hash Map'. In: *Proceedings of DISC*. 2017.

[115] Kyle J Nesbit and James E Smith. 'Data cache prefetching using a global history buffer'. In: *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE. 2004, pp. 96–96.

[116] Geraldo F Oliveira et al. 'DAMOV: A new methodology and benchmark suite for evaluating data movement bottlenecks'. In: *IEEE Access* 9 (2021), pp. 134457–134502.

[117] Ismail Oukid et al. 'FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-tree for Storage Class Memory'. In: *Proceedings of SIGMOD*. 2016.

[118] Ismail Oukid et al. 'Memory management techniques for large-scale persistent-main-memory systems'. In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1166–1177.

[119] Susan Owicki and Anant Agarwal. 'Evaluating the performance of software cache coherence'. In: *ACM SIGARCH Computer Architecture News* 17.2 (1989), pp. 230–242.

[120] Biswabandan Panda. 'Clip: Load criticality based data prefetching for bandwidth-constrained many-core systems'. In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023, pp. 714–727.

[121] R Hugo Patterson et al. 'Informed prefetching and caching'. In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. 1995, pp. 79–95.

[122] William Pugh. 'Skip Lists: A Probabilistic Alternative to Balanced Trees'. In: *Communications of the ACM* 33.6 (1990).

[123] Azalea Raad, Luc Maranget and Viktor Vafeiadis. 'Extending Intel-X86 consistency and persistency: Formalising the semantics of Intel-X86 memory types and non-temporal stores'. In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022), pp. 1–31.

[124] Pandian Raju et al. 'Pebblesdb: Building key-value stores using fragmented log-structured merge trees'. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 497–514.

[125] Viktor Rosenfeld, Sebastian Breß and Volker Markl. 'Query processing on heterogeneous CPU/GPU systems'. In: *ACM Computing Surveys (CSUR)* 55.1 (2022), pp. 1–38.

[126] Andy Rudoff, Chet Douglas and Tiffany Kasanicky. *Persistent Memory in CXL*. 'https://www.snia.org/sites/default/files/PM-Summit/2021/snia-pm-cs-summit-Rudoff-PM-in-CXL-2021.pdf'. 2022.

[127] Stephen M Rumble, Ankita Kejriwal and John Ousterhout. 'Log-structured Memory for DRAM-based Storage'. In: *12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*. 2014, pp. 1–16.

[128] Berk Saglam et al. 'Data prefetching on processors with heterogeneous memory'. In: *Proceedings of the International Symposium on Memory Systems*. 2024, pp. 45–60.

[129] Samsung. *Samsung Electronics Introduces Industry's First 512GB CXL Memory Module*. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module. 2024.

[130] Deeksha Satish and S Manimala. 'A STUDY ON CACHE REPLACEMENT POLICIES IN COHERENT CHIP MULTIPROCESSOR SYSTEMS (CMPs)'. In: *International Journal For Technological Research In Engineering, Volume 5* (2018).

[131] Gabin Schieffer et al. 'Harnessing Integrated CPU-GPU System Memory for HPC: a first look into Grace Hopper'. In: *arXiv preprint* (2024). arXiv:2407.07850.

[132] Gabin Schieffer et al. 'Harnessing integrated cpu-gpu system memory for hpc: a first look into grace hopper'. In: *Proceedings of the 53rd International Conference on Parallel Processing*. 2024, pp. 199–209.

[133] David Schwalb et al. 'nvm malloc: Memory Allocation for NVRAM.' In: *Adms@ Vldb* 15 (2015), pp. 61–72.

[134] ServeTheHome. 'The 2025 PCIe GPU in Server Guide'. In: *ServeTheHome* (2025). Accessed: 2025-09-25. URL: https://www.servethehome.com/the-2025-pcie-gpu-in-server-guide-supermicro-nvidia/.

[135] Debendra Das Sharma, Robert Blankenship and Daniel S Berger. 'An Introduction to the Compute Express Link (CXL) Interconnect'. In: *arXiv preprint arXiv:2306.11227* (2023).

[136] Ying Sheng et al. 'Flexgen: High-throughput generative inference of large language models with a single gpu'. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 31094–31116.

[137] Seunghee Shin, James Tuck and Yan Solihin. 'Hiding the long latency of persist barriers using speculative execution'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 175–186.

[138] Ben Simner et al. 'ARMv8-A system semantics: instruction fetch in relaxed architectures'. In: *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory*

and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings 29. Springer International Publishing. 2020, pp. 626–655.

[139] Yongju Song et al. 'Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices'. In: (2023).

[140] Alex Suhan and Todd Mostak. *MapD: Massive throughput database queries with LLVM on GPUs*. 2015.

[141] David Tam, Reza Azimi and Michael Stumm. 'Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors'. In: *ACM SIGOPS Operating Systems Review* 41.3 (2007), pp. 47–58.

[142] Yu Tang et al. 'DELTA: Memory-Efficient Training via Dynamic Fine-Grained Recomputation and Swapping'. In: *ACM Transactions on Architecture and Code Optimization* 21.4 (2024), pp. 1–25.

[143] Sardar Usman et al. 'Data locality in high performance computing, big data, and converged systems: An analysis of the cutting edge and a future system architecture'. In: *Electronics* 12.1 (2022), p. 53.

[144] Jianguo Wang and Qizhen Zhang. 'Disaggregated database systems'. In: *Companion of the 2023 International Conference on Management of Data*. 2023, pp. 37–44.

[145] Jing Wang et al. 'Pacman: An Efficient Compaction Approach for {Log-Structured}{Key-Value} Store on Persistent Memory'. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 773–788.

[146] Pengyu Wang et al. 'Grus: Toward unified-memory-efficient high-performance graph processing on gpu'. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 18.2 (2021), pp. 1–25.

[147] Tianzheng Wang, Justin Levandoski and Per-Ake Larson. 'Easy Lock-Free Indexing in Non-Volatile Memory'. In: *Proceedings of ICDE*. 2018.

[148] Zhendong Wang et al. 'Understanding and tackling the hidden memory latency for edge-based heterogeneous platform'. In: *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. 2020.

[149] Marcel Weisgut et al. 'CXL Memory Performance for In-Memory Data Processing'. In: *Proceedings of the VLDB Endowment* 18.9 (2025), pp. 3119–3133.

[150] Wikipedia contributors. *Lamport's bakery algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 19-February-2026]. 2026. URL: https://en.wikipedia.org/w/index.php?title=Lamport%27s_bakery_algorithm&oldid=1332615171.

[151] Henry Wong. *Intel Ivy Bridge Cache Replacement Policy*. https://blog.stuffedcow.net/2013/01/ivb-cache-replacement/. 2013.

[152] Carole-Jean Wu et al. 'PACMan: prefetch-aware cache management for high performance caching'. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 2011, pp. 442–453.

[153] Kai Wu et al. 'Ribbon: High performance cache line flushing for persistent memory'. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020, pp. 427–439.

[154] Kan Wu et al. '{NyxCache}: Flexible and efficient multi-tenant persistent memory caching'. In: *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 2022, pp. 1–16.

[155] Xiaoxiang Wu, Baptiste Lepers and Willy Zwaenepoel. 'Pre-Stores: Proactive Software-guided Movement of Data Down the Memory Hierarchy'. In: *Proceedings of the Twentieth European Conference on Computer Systems*. 2025, pp. 1161–1176.

[156] Fei Xia et al. 'HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems'. In: *Proceedings of USENIX ATC*. 2017.

[157] Lingfeng Xiang et al. 'Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering'. In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 488–505.

[158] Jian Xu and Steven Swanson. 'NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories'. In: *Proceedings of USENIX FAST*. 2016.

[159] Jian Xu et al. 'Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks'. In: *Proceedings of ASPLOS*. 2019.

[160] Yi Xu et al. 'Pie: Pooling cpu memory for llm inference'. In: *arXiv preprint arXiv:2411.09317* (2024).

[161] Yuanchao Xu et al. 'Asymmetry & Locality-Aware Cache Bypass and Flush for NVM-Based Unified Persistent Memory'. In: *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE. 2019, pp. 168–175.

[162] Jun Yang et al. 'NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems'. In: *Proceedings of FAST*. 2015.

[163] Juncheng Yang, Yao Yue and Rashmi Vinayak. 'Segcache: a memory-efficient and scalable in-memory key-value cache for small objects.' In: *NSDI*. 2021, pp. 503–518.

[164] Xinjun Yang et al. 'Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases'. In: *Companion of the 2025 International Conference on Management of Data*. 2025, pp. 689–702.

[165] Yujie Yang et al. 'Architectural and System Implications of CXL-enabled Tiered Memory'. In: *arXiv preprint arXiv:2503.17864* (2025).

[166] Ting Yao et al. 'MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM'. In: *Proceedings of USENIX ATC*. 2020.

[167] Seung Won Yoo et al. '{DJFS}:{Directory-Granularity} Filesystem Journaling for {CMM-H}{SSDs}'. In: *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 2025, pp. 35–51.

[168] Yichao Yuan et al. 'Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics'. In: *arXiv preprint arXiv:2502.09541* (2025).

[169] Bowen Zhang et al. 'NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems'. In: *Proceedings of the VLDB Endowment* 15.6 (2022), pp. 1187–1200.

[170] Feng Zhang et al. 'Understanding co-running behaviors on integrated CPU/GPU architectures'. In: *IEEE Transactions on Parallel and Distributed Systems* 28.3 (2016), pp. 905–918.

[171] Kai Zhang et al. 'Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores'. In: *Proceedings of the VLDB Endowment* 8.11 (2015), pp. 1226–1237.

[172] Wenhui Zhang et al. 'ChameleonDB: a key-value store for optane persistent memory'. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 194–209.

[173] Xiao Zhang, Sandhya Dwarkadas and Kai Shen. 'Towards practical page coloring-based multicore cache management'. In: *Proceedings of the 4th ACM European conference on Computer systems*. 2009, pp. 89–102.

[174] Yiying Zhang and Steven Swanson. 'A study of application performance with non-volatile main memory'. In: *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2015, pp. 1–10.

[175] Yongkang Zhang et al. 'Missile: Fine-Grained, Hardware-Level GPU Resource Isolation for Multi-Tenant DNN Inference'. In: *arXiv e-prints* (2024), arXiv–2407.

[176] Yuhong Zhong et al. 'Managing Memory Tiers with {CXL} in Virtualized Environments'. In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 37–56.

[177] Yuzhi Zhuang, Ming Zhang and Binghao Wang. 'A Generalized Optimization Scheme for Memory-Side Prefetching to Enhance System Performance'. In: *Electronics* 14.14 (2025), p. 2811.

[178] Sergey Zhuravlev, Sergey Blagodurov and Alexandra Fedorova. 'Addressing shared resource contention in multicore processors via scheduling'. In: *ACM Sigplan Notices* 45.3 (2010), pp. 129–142.

[179] Pengfei Zuo, Yu Hua and Jie Wu. 'Write-optimized and High-performance Hashing Index Scheme for Persistent Memory'. In: *Proceedings of OSDI*. 2018.

[180] Yoav Zuriel et al. 'Efficient Lock-free Durable Sets'. In: *Proceedings of OOPSLA* (2019).