UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EFFICIENT AND SCALABLE ARCHITECTURES FOR PERSISTENT MEMORY SYSTEMS

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

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Matheus Ogleari

June 2019

The Dissertation of Matheus Ogleari
is approved:

Professor Jishen Zhao, Co-Chair

Professor Ethan Miller, Co-Chair

Professor Matthew Guthaus

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Efficient and Scalable Architectures for
Persistent Memory Systems

by

Matheus Ogleari

Memory systems are evolving in a multitude of ways in state-of-the-art computer systems and in computer architecture. The changes range from the low-level underlying technology used to implement memory, to the high-level ways in which memory systems are structured and organized. These changes have tremendous implications on the fundamental performance of computer systems because of how integral a role memory plays in the design of these systems. Traditionally, the memory hierarchy has been the bottleneck for how well computer systems perform and the resources they consume. Computer architects face significant challenges in meeting the growing memory demands and efficiently integrating these new technologies. Computer architecture must adapt to these changes in a high-performance, cost-effective way. Modern problems require modern solutions.

This thesis proposes efficient architecture solutions for two primary contemporary developments in memory: persistent memory and memory networks. Persistent memory is a subset of nonvolatile memory with similar properties of storage systems. As such, it comes with not only many benefits, but also a number of challenges inherent to both storage systems but to new technologies in general. Similarly, memory networking shares similar properties to network on-chip as well as computer networking design. Persistent memory and memory networks can go hand in hand to implement high performance computing or database systems to meet the demands of current applications.

First this thesis tackles persistent memory and proposes a persistency-aware prefetcher design, eFetch. This design considers redo and undo logging, a commonly used technique in persistent memory systems. It capitalizes on memory access patterns and commonly used prefetching techniques to improve system performance. Development and analysis of this design motivates this thesis to additionally investigate cacheable logging and the balance between software and hardware in addressing these challenges. Across various microbenchmarks, this

design achieves up to 1.33× IPC improvement, 1.24× higher throughput, a 1.39× reduction in dynamic memory energy consumption, and a 1.36× decrease in overall memory traffic.

Second, this thesis proposes an efficient undo+redo logging hardware mechanism for persistent memory that accommodates more recent advancements in persistent memory. It better leverages the existing underlying hardware. It also explores implementation in RISC-V and enables future research and hardware evaluation and experimentation in persistent memory. Evaluating this using both real workloads as well as microbenchmarks shows a significant improvement in system throughput (up to 2.7×). It also reduces both dynamic energy (by up to 2.43×) and memory traffic.

Finally, this thesis addresses memory networks and the current problems in contending with increasing memory application and server demands. This thesis establishes three primary features for memory networks: scalability, arbitrary scale, and elasticity. As such, this thesis proposes String Figure, a memory network design and topology that attains three of these goals while still performing well against baseline or alternative memory network designs. String Figure competes with other memory network designs and out performs them in a number of network traffic patterns and real workloads.

Brief Biography

Matheus Almeida Ogleari was born on April 28, 1991 in Salvador, Brazil. He is the son of Arnaldo Ogleari and Gilka Ogleari. He has one older brother, Tamir Almeida. When Matheus was three years old, he and his family immigrated to the United States for a better life. Matheus and his family moved around a lot and lived in Florida, Oklahoma, and Pennsylvania.

At a young age, Matheus got into computers and soon became the go-to IT guy for his family whenever they had technology problems. His interest in video games fueled this. By 1998, Matheus was on a Windows 98 computer playing card games with people over the internet and participating and moderating on IRC chat rooms, learning input commands and modifying scripts on the platform. In middle school, he began dabbling in game design and modifying maps and scripts for his favorite game at the time, Warcraft 3. This grew into an early interest in programming by the beginning of high school. Matheus attended Cornell University for his undergraduate studies where he majored in electrical and computer engineering with a minor in computer science. During the program, Matheus quickly became enamored with computer architecture. Matheus dabbled both with academic research via his summer 2011 research with Professor Christopher Batten, and with industry via his summer 2012 internship at Intel. His positive experience with Professor Batten and his group encouraged him to apply to graduate school. Matheus began his PhD at the University of California, Santa Cruz. He eventually joined the lab of his advisor, Professor Jishen Zhao. Under her guidance, Matheus learned about persistent memory and began exploring architecture solutions and projects that aligned with his interests. He also gained invaluable experience through collaboration with other research groups, as well as through his internships at Intel, Huawei, and Western Digital.

In his free time, Matheus enjoys playing board games and video games, watching movies, discussing politics, traveling, enjoying local cuisine, and spending time with friends. His favorite video games include Fallout: New Vegas, League of Legends, Super Smash Bros Ultimate, and The Legend of Zelda: Breath of the Wild. His favorite board games include Cosmic Encounter, Spirit Island, Terraforming Mars, Twilight Struggle, and Star Wars: Rebellion. Matheus also has a passion for teaching. One day, he hopes to travel the world visiting many countries and eventually settle down to pursue one of his hobbies. Matheus is very grateful for his PhD experience and all those who were a part of it. It has been an unforgettable and indispensable experience.

I dedicate this to my family, the source of inspiration and motivation in my life and whom I always want to make proud.

Acknowledgments

I would like to acknowledge the tremendous contributions of my advisor and other professors I've worked with over the years as well as my sources of inspiration for pursuing my PhD degree. The technical contributions with more detailed acknowledgments of other collaborators are in Section 1.4. I want to use this section for personal thanks to them as well as the lasting friendships I've made as a graduate student.

I would like to thank Professor Christopher Batten at Cornell University for taking me under his wing when I was an undergraduate student. My experience doing research for him in his research group helped propel me into applying for a graduate PhD program. I also want to thank all the students and colleagues at Cornell and in Professor Batten's group I've worked with or admired. Aadeetya Shreedhar, one of my longest and closest friends, I thank for always being there for me when I needed him, and for all the projects we worked on together over many long nights. Sean Clark, I thank for being friendly and approachable, and for joining me in exploring research as an undergraduate. Ji Kim, Derek Lockhart, and Shreesha Srinath, Professor Batten's graduate students at the time, I thank for being role models of strong character and impeccable work ethic. They were inspiring examples of the graduate student I hoped to be one day.

I would like to thank Professor Jose Renau for his help and accepting me into the computer engineering PhD program at the University of California, Santa Cruz. I would like to thank all the students in Professor Renau's lab for being accepting and friendly. Ehsan Ardestani, I thank for the opportunity to TA in his class as well as for the internship opportunity in his group at Huawei. Daphne Gorman, I thank for her continued friendship. Ramesh Jayaraman, I thank for his close friendship as well as for being a great reliable housemate for a large part of my PhD program.

I would like to thank my mentor and advisor, Professor Jishen Zhao, for accepting me into her lab and guiding me through the toughest years of my PhD. She's helped me mature as both a researcher and a person. I thank her for pushing me through some of my lowest, most hopeless moments as well as through my happiest times. I want to continue to strive to be as hard working as she is. I appreciate her and all her efforts, my time in her research group has been unforgettable.

I want to thank all the students with whom I worked in Jishen's group. I thank Xiao Liu, for being a good dependable friend and for being a good listener and support during hard

times. I thank Mengjie Li for his friendship and help in administering the lab at UC Santa Cruz. I thank Hengyu Zhao and Zixuan Wang for being good company in our lab.

I would like to thank Professor Ethan Miller, for giving me a place in his research group, providing funding, and for accommodating me and handling my administrative needs after my lab moved to UC San Diego. I also want to thank him for his indispensable contribution and insight for a number of the underlying ideas in my research. I also want to thank SSRC, the Storage Systems Research Center at UCSC for giving me a platform and supporting my research efforts through their effective resources. I would like to thank my other research collaborators as well: Xing Hu, Professor Chen Qian, and Ye Yu.

I would like to thank Professor Matthew Guthaus for being part of my PhD thesis, as well as advancement committee. I also thank him for his affability and wisdom at times when I needed it. I want to thank Riadul Islam, from Professor Guthaus's lab, for being a good friend and fellow TA, as well as for helping to guide me through some of the nuances and logistical matters of the PhD program.

I would also like to thank Intel, Huawei, and Western Digital, the three companies that offered me internships during my PhD program. The industry experience I gained working at these companies and the exposure to being an engineer were crucial and kept me motivated to continue and complete my studies.

Last but definitely not least, I would like to thank my family for their love, support, and patience. Since I was young, my parents, Arnaldo and Gilka Ogleari, instilled in me a sense of importance in education. My older brother, Tamir Almeida, mentored me closely and guided me through the college application process. He shared with me his wisdom and experience I needed to help steer my academic career choices. I hope I go on to make them proud in my career and beyond.

In terms of funding, this thesis was supported in part by NSF grants 1652328, 1718158, 1829524, 1829525, NSF I/UCRC Center for Research on Storage Systems, SRC/DARPA Center for Research on Intelligence Storage and Processing-in-memory, and the SSRC Storage Systems Research Center.

Chapter 1

Introduction

The field of computer architecture has yielded tremendous achievement in advancing and improving technology for a large variety of applications and uses in daily life. With Moore's Law coming to its end, architects are increasingly resorting to alternate ways to improve performance of computer systems. These alternatives include hardware acceleration and chip specialization. These are able to achieve high performance and energy efficiency by directly targeting the specific application or class of applications. This is substantially more important with the advent of newer memory technologies. Memory systems have often been the bottleneck in the costs and performance of computer systems. The biggest question is how much improvement and optimization architects can exact from improvements to hardware. The most optimal solutions require the deepest levels of understanding of the underlying microarchitecture.

1.1 Persistent Memory

Persistent memory presents a new tier of data storage components for future computer systems. By attaching Non-Volatile Random-Access Memories (NVRAMs) [24, 59, 124, 150] to the memory bus, persistent memory unifies memory and storage systems. NVRAM offers the fast load/store access of memory with the data recoverability of storage in a single device. Consequently, hardware and software vendors recently began adopting persistent memory techniques in their next-generation designs. Examples include Intel's ISA and programming library support for persistent memory [57], ARM's new cache write-back instruction [9], Microsoft's storage class memory support in Windows OS and in-memory databases [29, 36], Red Hat's

persistent memory support in the Linux kernel [99], and Mellanox's persistent memory support over fabric [35].

Though promising, persistent memory fundamentally changes current memory and storage system design assumptions. Reaping its full potential is challenging. Previous persistent memory designs introduce large performance and energy overheads compared to native memory systems, without enforcing consistency [52, 102, 146]. A key reason is the write-order control used to enforce data persistence. Typical processors delay, combine, and reorder writes in caches and memory controllers to optimize system performance [54, 73, 109, 146]. However, most previous persistent memory designs employ memory barriers and forced cache write-backs (or cache flushes) to enforce the order of persistent data arriving at NVRAM. This write-order control is sub-optimal for performance and does not consider natural caching and memory scheduling mechanisms.

1.1.1 Prefetching

Most previous persistent memory designs focus on optimizing software-based logging algorithms to reduce the overhead of data persistence [30] [135] [89]. Several recent studies strive to reduce such overhead by optimizing hardware-based versioning and write-order control mechanisms [146] [64] [85] [109] [148] [116] [73]. Yet the data persistence property obviates many basic assumptions of current memory architecture design, leading to significant degradation performance and energy efficiency. In particular, this thesis observes that the hardware prefetcher designs employed in commodity processors can substantially increase conflict misses in CPU caches when executing persistent memory applications. This is because traditional prefetchers do not distinguish between the working data and the log. They tend to over-prefetching the non-reusable log blocks, which can kick the working data out of caches.

Moreover, this thesis identifies that substantial opportunities exist in exploiting intelligent hardware prefetcher design to optimize persistent memory performance. In particular, prefetching can be leveraged to parallelize and merge the memory requests to perform logging and working data updates. As a result, unnecessary memory access and data movement can be significantly reduced.

These issues and opportunities manifest in the use of persistent memory that architects must rethink hardware prefetcher design to reap the full potential of persistent memory

technique. As such, this thesis proposes rethinking the hardware prefetcher design. This thesis proposes a persistence-aware prefetcher design, eFetch[1], which consists of persistent write tracking, redo log prefetching, and undo logging triggered prefetching mechanisms. eFetch adopts a special log address buffer, which stores the cache tags associated with the log memory addresses. Such memory address tracking prevents the over-prefetching that can stretch to the log. Furthermore, this thesis proposes two intelligent prefetching mechanisms, which leverage prefetching to i) relax the ordering constraints between persistent memory access and ii) parallelize logging and working data updates in redo and undo log based systems, respectively.

1.1.2 Hardware Undo+Redo Logging

Several recent studies strive to relax write-order control in persistent memory systems [73, 109, 146]. However, these studies either impose substantial hardware overhead by adding NVRAM caches in the processor [146] or fall back to low-performance modes once certain bookkeeping resources in the processor are saturated [73].

A **goal** of this thesis is to design a high-performance persistent memory system without (i) an NVRAM cache or buffer in the processor, (ii) falling back to a low-performance mode, or (iii) interfering with the write reordering by caches and memory controllers. The key idea is to maintain data persistence with a combined undo+redo logging scheme in hardware.

Undo+redo logging stores both old (undo) and new (redo) values in the log during a persistent data update. It offers a key benefit: relaxing the write-order constraints on caching persistent data in the processor. This thesis shows that undo+redo logging can ensure data persistence without needing strict write-order control. As a result, the caches and memory controllers can reorder the writes like in traditional non-persistent memory systems (discussed in Section 3.1.2).

Previous persistent memory systems typically implement *either* undo or redo logging in software. However, high-performance software undo+redo logging in persistent memory is infeasible due to inefficiencies. First, software logging generates extra instructions in software, competing for limited hardware resources in the pipeline with other critical workload operations. Undo+redo logging can double the number of extra instructions over undo or redo

[1] A system with nonvolatile main memory may also employ volatile DRAM, controlled by a separate memory controller. This work focuses on the case where persistent data updates need to be maintained in nonvolatile main memory, which is a common case in database and file system applications.

logging alone. Second, logging introduces extra memory traffic in addition to working data access [146]. Undo+redo logging would impose more than double extra memory traffic in software. Third, the hardware states of caches are invisible to software. As a result, software undo+redo logging, an idea borrowed from database mechanisms designed to coordinate with software-managed caches, can only conservatively coordinate with hardware caches. Finally, with multithreaded workloads, context switches by the operating system (OS) can interrupt the logging and persistent data updates. This can risk the data consistency guarantee in multi-threaded environment (Section 3.1.3 discusses this further).

Several prior works investigated hardware undo or redo logging separately [73, 97] (Section 5.2). These designs have similar challenges such as hardware and energy overheads [97], and slowdown due to saturated hardware bookkeeping resources in the processor [73]. Supporting both undo and redo logging can further exacerbate the issues. Additionally, hardware logging mechanisms can eliminate the logging instructions in the pipeline, but the extra memory traffic generated from the log still exists.

To address these challenges, this thesis proposes a combined undo+redo logging scheme in hardware that allows persistent memory systems to relax the write-order control by leveraging existing caching policies. The design consists of two mechanisms. First, a **Hardware Logging (HWL)** mechanism performs undo+redo logging by leveraging write-back write-allocate caching policies [54] commonly used in processors. The HWL design causes a persistent data update to automatically trigger logging for that data. Whether a store generates an L1 cache hit or miss, its address, old value, and new value are all available in the cache hierarchy. As such, this design utilizes the cache block writes to update the log with word-size values. Second, this thesis proposes a cache **Force Write-Back (FWB)** mechanism to force write-backs of cached persistent working data in a much lower, yet more efficient frequency than in software models. This frequency depends only on the allocated log size and NVRAM write bandwidth, thus decoupling cache force write-backs from transaction execution.

1.2 Memory Networks

The volume of data has skyrocketed over the last decade, growing at a pace comparable to Moore's Law [96]. This trend drives the popularity of big data analytics [19, 120], in-memory computing [45,137], deep learning [121,131,132], and server virtualization [19], which

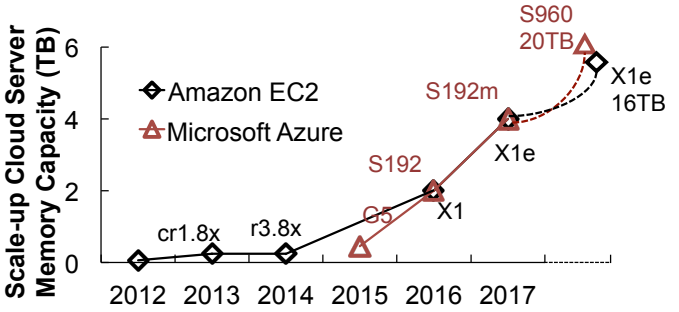


Figure 1.1: Expanding server memory demand.

are frequently insatiable memory consumers. As a result, these applications demand a continuous expansion of server memory capacity and bandwidth to accommodate high-performance working data access. As shown in Figure 1.1, cloud server memory capacity has been rapidly growing since the debut of cloud service for in-memory computing [4, 20, 119].

Unfortunately, DRAM capacity scaling falls far behind the pace of the application demand with current DDRx based architectures [100]. One conventional solution to increasing server memory capacity is to add more CPU sockets to maintain additional memory channels. In commodity systems, each processor can support up to 2TB memory. As a result, EC2 1Xe adopts four CPU sockets to accommodate the 4TB memory capacity [20]. Azure S960 servers can adopt 20 Intel Xeon processors to accommodate 20TB of memory [4]. However, purchasing extra CPU sockets substantially increases the total system cost [144, 145]. The extra hardware cost adds significant, often nonlinear overheads to the system budget, making this solution unsustainable.

A promising technique to tackle these memory challenges is memory networking, whereby the server memory system consists of multiple 3D die-stacked memory nodes interconnected by a high-speed network. The interconnected memory nodes form a disaggregated memory pool shared by processors from different CPU sockets in the server (Figure 4.1). Ideally, the memory network can enable more scalable performance and capacity than traditional DDRx-based memory systems, as shown by academia [67, 112, 144] and industry efforts from HPE [8] and IBM [130]. However, the memory network scalability relies on the scalability of the memory fabric that interconnects the memory nodes [112, 144].

Previous memory network designs investigated platforms with limited numbers of memory nodes. Scalability and flexibility were not considered in their the design goals. Given

that each memory node (a 3D memory stack) can offer 8GB capacity [144], state-of-the-art 4TB server memory system requires 512 memory nodes. Recent works have proposed optimizing NoC network topologies to support up to 128 memory nodes in integrated CPU+GPU (APU) systems [112]. However, the design partitions clusters of memory nodes to separate channels, where each processor can only access a subset of the memory space. State-of-the-art planar topologies [69, 70] offer high network throughput at a large scale. However, the number of required router ports and links increases as the network size grows, which imposes undesirable cost in routers integrated with memory nodes. The challenges of scaling up server memory capacity still remain.

A **goal** of this thesis is also to design a high-performance, scalable, and flexible memory network architecture that can support over a thousand interconnected memory nodes shared by processors in a cloud server. This memory network is designed around three primary design principles: scalability, arbitrary network sizing, and elasticity. These are discussed further in depth.

To achieve this, this thesis proposes String Figure, a scalable and elastic memory network architecture that consists of three design components. First, this thesis proposes a network construction scheme to efficiently generate random network topologies that interconnect an arbitrary number of memory nodes with high network throughput, near-optimal path lengths, and limited router ports. Second, this thesis develops an adaptive greedy routing protocol, which significantly reduces the computation and storage overhead of routing in each router. Finally, this thesis proposes a network reconfiguration scheme, which allows the network scale, topology, and routing to change according to power management and design reuse requirement. Performance and energy experiments show that String Figure can interconnect up to 1296 memory nodes with significantly higher throughput and energy efficiency than previous designs, across various of synthetic and real workloads.

1.3 Thesis Overview

This thesis proposes three designs: eFetch, undo+redo hardware logging, and String Figure. This thesis goes in-depth on these designs, elaborating on the necessity of each in sequential chapters. At the end of each chapter, it also explains how these designs relate to each other and how this bridge was formed over the course of the research. In particular, the

sequence of these three designs also represents the growing scale and breadth of challenges associated with new memory technologies and techniques. EFetch starts with a system focusing on one aspect of a processor with persistent memory (a prefetcher). With hardware undo+redo logging, the entire system is considered with how best to deal with persistent memory accesses. With String Figure, the thesis expands to large scale memory systems and the interconnection of various nodes and memory subsystems across vast networks.

Chapter 2 details and describes the design for eFetch as well as the baseline assumptions for persistent memory. Despite its breadth of research and underlying foundations, there is still debate about what these assumptions should be. For example, should persistent memory logs be cached? If not, how does a processor deal with uncacheable logs? Is this the most efficient way? How do other units within the processor deal with logging, a technique so commonly used in persistent memory? EFetch presents an early design for understanding and exploring the design space for persistent memory, and addressing the challenges that manifest in the performance gap between persistent and traditional memory systems.

Chapter 3 describes hardware undo+redo logging and its design. It represents the next step in evolution from eFetch. It more fully encompasses the entire chip and shifts persistent memory design focus less on software and more toward hardware. This chapter justifies this choice and presents an efficient and cost effective way to achieve this. It also further advocates for assumptions made from eFetch about the underlying persistent memory system and the implications for future work. In particular, it strives to expand persistent memory research into modern processors and computer systems like RISC-V. Additionally, this chapter brushes on additional requirements for persistent memory and changes needed in hardware when dealing with RDMA requests over networks to persistent memory.

Chapter 4 describes the design for String Figure, a scalable and elastic memory network design. This design was motivated by the necessity of optimally handling persistent memory requests across networks via RDMA or other paradigms. String Figure takes this a step further, offering a topology and set of network design goals that meet the demands for large scale memory networks that are not just persistent or for persistent applications. The performance gains and features offered by String Figure benefits all memory systems, persistent or otherwise. It also

Chapter 5 catalogs the vast body of related work on prefetching, persistent memory,

and memory networks. It solidifies the novelty and impact of the designs proposed in this thesis. Chapter 6 summarizes and elaborates on the contributions of this thesis, highlighting the potential for future work and research in these topics.

The primary contributions of this thesis are condensed as follows:

- This thesis identifies new performance and energy issues associated with current persistent memory systems and how persistent-aware prefetching can reduce data movement across the memory bus.
- This thesis enables efficient undo+redo logging for persistent memory systems in hardware using lightweight software support and low complexity processor modifications. This relaxes write ordering constraints and addresses the challenges inherent to using both undo and redo logging together.
- This thesis constructs a memory network topology and design that is able to scale with very large number of nodes, support any number of nodes in its network without performance loss, and supports adding or removing nodes from its network at low cost without having to rebuild the entire system.

1.4 Collaboration, Funding, and Previous Publications

This thesis would not have been possible without the immense effort and dedication of all those involved in my publications. This includes professors as well as graduate and post-graduate students with whom I have had the pleasure of working. My advisor, Jishen Zhao, was a key collaborator in all my work by helping to iteratively evolve the ideas and key concepts behind the designs described in this paper. Her experience in persistent memory systems attributes to this. The framework from her previous work in persistent memory helped propel my own research. She also helped me organize and refine the narrative and how it is presented in this thesis. Professors Ethan Miller and Chen Qian at University of California, Santa Cruz provided invaluable insight and basis for the primary designs described in this thesis. Professor Miller's expertise in storage systems and Professor Qian's expertise in networking were strong foundations in this regard.

Collaborators from my and other research groups are also responsible for making this thesis possible. Mengjie Li in my group was a great resource in helping to study and learn

the effects of cacheable versus uncacheable logging in persistent memory. My work with Xing Hu at University of California, Santa Barbara sparked my interest in exploring RDMA and network design as another side of improving not only persistent memory performance but large memory systems in general. Ye Yu from the University of Kentucky (now Google) provided a starting framework from his Space Shuffle design that directly helped the development of String Figure. My exposure and contribution to work on processing-in-memory with Hengyu Zhao of my group emphasized the memory demands of modern applications for me which motivates String Figure. My invaluable internship opportunity at Western Digital thanks to my manager Martin Lueker-Boden opened up my work in persistent memory to RISC-V support and design exploration. The internship gave me access to industry-standard tools (like the Xilinx Zynq zc706 FPGA), support, and experience of an engineering company. Co-workers Tung Hoang, Pi-feng Chiu, and Christoph Hellwig helped guide me in this.

Two previous publications have contributed significantly to this thesis. My work on undo+redo logging [104] and String Figure [105] are the basis for Chapters 3 and 4, respectively. My collaboration with Xing Hu on Persistence Parallelism Optimization [55] contributed to the RDMA network discussion in Chapter 3. My collaboration with Mengjie Li on Logging in Persistent Memory [79] motivated and led to many design decisions throughout this thesis, discussed more in Chapter 2.

Chapter 2

Prefetching and Logging in Persistent Memory

Persistent Memory functions as a hybrid of traditional storage systems with main memory. It combines the benefits of both worlds – the persistence of storage disks with the byte-addressability and load-store interface of main memory. It improves upon the emerging nonvolatile memory (NVRAM) technologies by persisting data with full integrity even during crashes and power outages. Yet, persistent memory systems impose additional memory accesses to maintain data persistence (a traditional storage property) in memory, by employing multiversioning mechanisms (e.g., logging) or copy-on-write mechanisms inherited from disk-based storage systems, such as databases and file systems. This chapter observes that such additional memory demand causes increased conflict misses with prefetched data in the cache, substantially hurting the performance of persistent memory applications. Furthermore, intelligent hardware prefetching can offer substantial opportunities in relaxing the performance constraint of persistent memory systems.

This the goal in this chapter is to achieve high-performance and energy-efficient hardware prefetching in persistent memory systems. This thesis proposes an efficient persistence-aware hardware prefetching mechanism, eFetch, which mitigates the performance and energy loss in persistent memory applications while leveraging prefetching to relax the ordering constraints in log-based persistent memory systems. Evaluation across various persistent memory applications demonstrates that this design leads to 1.33× IPC improvement, 1.24× higher

throughput, a 1.39× reduction in dynamic memory energy consumption, and a 1.36× decrease in overall memory traffic.

2.1 Background and Prefetching

Persistent memory bridges the the benefits of memory and storage. It allows data to be persisted directly through a fast load/store memory interface, obviating the performance and energy overhead of accessing slow storage components such as disks and flash. As such, systems incorporating persistent memory can offer promising data storage solutions for various computer systems, whether they be cloud systems, embedded, or mobile, which are expected to generate, process, and store increasingly large volumes of data despite operating with constrained voltage/thermal margins or unreliable power sources.

2.1.1 Supporting Data Persistence in Memory

Persistent memory systems need to protect data integrity against partial or reordered writes to nonvolatile main memory (NV memory), despite crashes and power outages. Taking an example of inserting a node in a linked list stored in NV memory, write-back caches and memory controllers can reorder the writes; eventually, the pointer can be written into NV memory before the node data. If the system loses power when the node data still remains in volatile CPU caches, the pointer becomes a dangling pointer and retains in NV memory across system reboots. As such, the linked list is broken and the inconsistent state will remain from that point on. To avoid this, persistent memory systems perform versioning and write-order control to ensure data persistence in memory.

Versioning can be implemented by two commonly-used methods: logging [135] [30] and copy-on-write [32]. Among the two, logging is an essential component in most database and file systems that are likely to run on persistent memory. Therefore, eFetch focuses on improving the performance and energy efficiency of log-based persistent memory systems.

Logging can be implemented as *redo logging* [135] [30], *undo logging*, or a combination of both (i.e., *undo+redo logging*) [95] [51]. Redo logging writes new data updates, along with their addresses, into a log before updating their original locations. If system loses power before logging is completed, persistent memory can always recover, using the intact original

data in memory. Some systems can employ undo logging, which instead writes old values of data into the log and updates the original data with new values. Many database systems adopt undo+redo logging [95] [51]. For example, ARIES [95] combines a physical redo logging with a logical undo logging process across transactions to ensure that the database can be recovered after crashes. Yet performing undo+redo logging through the memory bus can introduce substantial performance overhead on top of the fast load/store interface. Therefore, most existing persistent system designs adopt either undo or redo logging. Most previous logging schemes can be categorized as write-ahead logging (WAL), e.g., they commit log records before updating corresponding original data. Recent studies show that write-behind logging (WBL), which updates data before committing to the log, can improve persistent memory performance in certain scenarios [44] [16].

In most previous persistent memory designs, the log is supposed to be used during system recovery; once a log entry is updated, it is not reused throughout applications execution. Therefore, most previous persistent memory systems do not cache the log. Write-order control enforces the order of log and original data updates arriving at NV memory. Otherwise, both types of updates can be simultaneously issued to NV memory; system crashes can interrupt these writes and leave both the log and the original data partially updated. Write-order control is enforced by a memory barrier between the writes to the log and writes to the original locations in memory. Such memory barriers are implemented by uncacheable writes or cache flush and memory fence instructions, such as `clflush`, `clwb`, `mfence`, and `pcommit`.

2.1.2 Hardware Prefetching

A hardware prefetcher is a hardware component used in microprocessors to speed up program execution by reducing the average memory access latency [39]. Modern microprocessors run much faster than the main memory. Therefore, main memory accesses typically lead to stalled cycles on the microprocessor. Prefetching occurs when a processor requests data or instructions from main memory before it is actually needed. Once a prefetched data or instruction comes back from memory, it is placed in the cache until it is accessed; reading the data or instruction from caches is much faster than if the microprocessor had to request it from main memory [46]. Yet the cache becomes populated more quickly and thus are more susceptible to *conflict misses*. This issue leads to a loss of overall performance by increasing the average

memory access latency (AMAL).

Current microprocessor designs adopt various hardware prefetching schemes, such as next-line prefetching, stride prefetching [46], locality-based prefetching [125], and Markov prefetching [62]. These are designed for conventional memory systems without data persistence support. This thesis studies the issues with traditional hardware prefetcher design in persistent memory systems and offers persistence-aware prefetching mechanisms that can work with various previous prefetcher designs.

2.2 The Case for Prefetching

This section discusses the reasons and motivation for pursuing performance speedups in prefetching for persistent memory systems.

The premise of prefetching is to generate values to populate the cache sooner, which can lead to additional cache hits (hence greater performance) in following memory accesses. Toward this end, most traditional hardware prefetchers exploit spatial locality by prefetching the addresses near the desired ones. Persistent memory systems typically allocate the log and temporarily data buffer in one or more contiguous regions in the physical address space. As a result, once the first data block of a log entry is accessed, hardware prefetchers can prefetch the subsequent data blocks. While persistent memory logs are not supposed to be cached (discussed in Section 2.1.1), hardware prefetchers can bring logs into caches. This thesis refers to this issue as over-prefetching of log data.

There are many prefetching algorithms designed for specific use cases or that target sets of applications. Prefetching policies are diverse and no single strategy has yet been proposed which provides optimal performance in all cases [133]. New prefetching techniques emerge with new technology and applications. Much how the emergence of newer applications with different memory access patterns have driven development of these techniques, so too does persistent memory. Persistent memory provides a new kind of predictable memory access pattern that can be used in new or existing code. Given this predictability, this thesis develops a prefetcher scheme fitting for these persistent applications.

The log is typically uncacheable because it has no reuse (Section 2.1.1); subsequent reads of the same data will need to wait until the working data is updated (shown in Figure 2.3(a) (the example code without redo log prefetching)). Yet new data values are actually already per-

sistent after redo log updates are completed. Therefore, the ordering constraint between working data updates and subsequent reads can be relaxed by allowing proactive reads from the redo log. However, redirecting the reads to the redo log can substantially complicate persistent memory design. As described in Section 2.3, eFetch allows such proactive reads without redirecting them.

Hardware Prefetching vs. Software Prefetching. Prior works have studied the differences between hardware prefetching and software prefetching [28]. This study and [152] cite that software prefetching uses compile-time information to do complex analysis. However, hardware prefetching has the advantage of dynamic information. While both improve miss rates in caches through prefetching, software prefetching actually leads to a higher overhead cost. These types of study have laid the foundation for modern prefetching research and prefetcher designs. The policies used are typically implemented in hardware and are lightweight. This thesis uses the same motivation for pursuing hardware-based prefetching for the eFetch design.

2.3 eFetch Design Overview

Overview. This thesis proposes eFetch, a hardware prefetcher design that addresses the aforementioned over-prefetching issue and exploits the opportunity of relaxing the ordering constraint. This prefetching scheme can be used in persistent memory systems that adopt undo logging, redo logging, or both to enforce data persistence. EFetch introduces three novel design principles to achieve this goal. First, *persistent write tracking* ensures that the hardware prefetcher can distinguish the addresses of working data and log updates, such that it can employ different prefetching policies for the two. Persistent write tracking effectively reduces the aforementioned over-prefetching issue. Second, *redo log prefetching* leverages prefetching of the latest redo log updates to simultaneously accommodate working data updates and enable proactive reads. Third, *undo logging triggered prefetching* exploits the fact that working data updates typically follow log updates to proactively starting working data updates in caches. The redo log prefetching and undo logging triggered prefetching mechanisms leverage prefetcher's support to relax the ordering constraint on persistent memory access; the two mechanisms also stretches the persistent memory performance by reducing redundant data movement across the memory bus. The realization of these three principles improves system performance and reduces system energy consumption as evaluated in the results. Furthermore, eFetch also

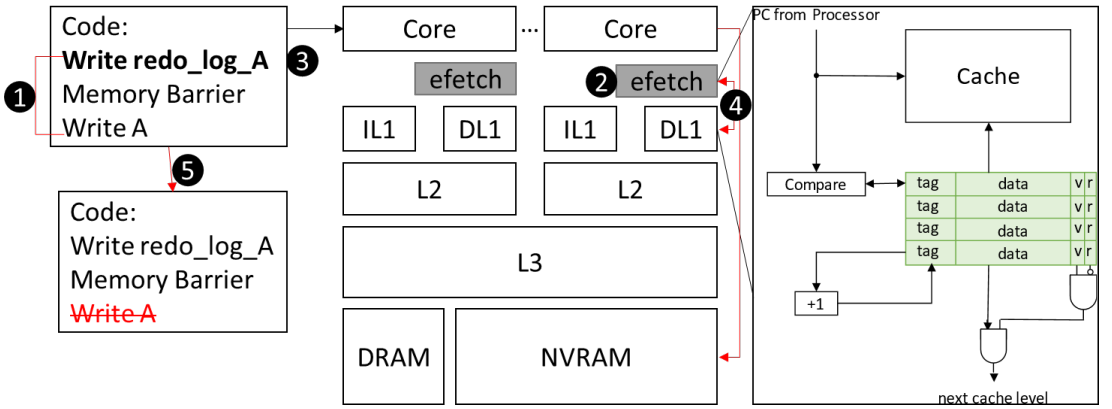


Figure 2.1: High-level overview of eFetch and how it interacts with the rest of the system and the application.

monitors the NV memory bandwidth utilization and prevents prefetching to interference with ordinary memory requests. Figure 2.1 depicts eFetch configuration. A step-by-step overview of how all mechanisms work together will be presented in Section 2.3.4.

2.3.1 Persistent Write Tracking

The goal to prevent traditional prefetching policies from prefetching non-reusable log updates. To achieve this, this thesis allows hardware prefetchers to identify the addresses of log updates. To this end, eFetch keeps track of the starting addresses of log updates by buffering the starting and the end addresses (virtual or physical addresses determined by which is used by the prefetcher) of each allocated log regions in NV memory. When the prefetcher initiates prefetching requests guided by traditional prefetching policies (e.g., next-line, stride, locality-based, Markov, etc.), it performs an "address check" against the buffered addresses. If the requesting address falls into a memory region for log, the prefetcher skips that request. Figure 2.2 summarizes this mechanism.

Making the Log Uncacheable. To comply with the persistence model described in Section 2.1, there first needs to be a mechanism to allow detecting memory accesses into the log to prevent them from caching in the original request. To this end, this design employs a software interface (discussed in Section 2.4) that distinguishes log accesses from other memory accesses. This software interface is similar to the ones adopted in previous persistent memory system designs [135]. This way, the existence and location of the logs are known at compile-time. It

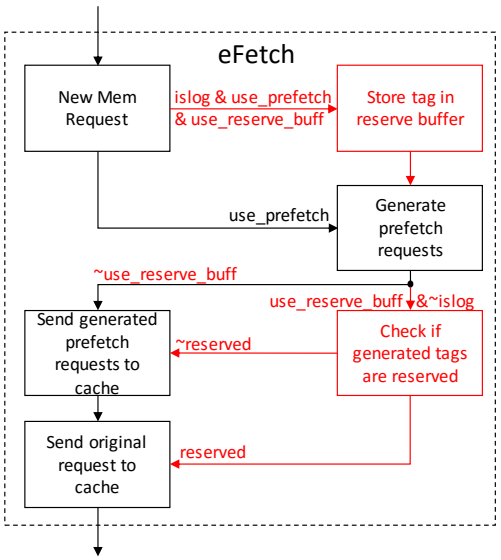


Figure 2.2: High-level flow chart for eFetch. Red highlighted regions shows added eFetch functionality atop traditional hardware prefetcher design.

can pass along this information to the eFetch hardware. As such, load and store instructions associated with accessing the log will be flagged to be treated differently than working data access, bypassing the cache hierarchy. This prevents the log access from polluting the cache with additional conflict misses.

Address Tracking. For the log to be uncacheable, the system needs to have the knowledge about its address in the memory space to prevent the cache from normally sending requests. This same mechanism can be used to identify when eFetch triggers. This trigger expands to the prefetcher to pass along the information of known log memory requests and the subsequent areas in memory. There can exist memory regions that share spatial locality with the log but not much else. Efetch does not prefetch these adjacent addresses because they tend to be isolated from the rest of the code, more specifically, from the desired persistent data structure. Furthermore, the prefetcher is also aware of memory accesses preceding the log that could trigger any over-prefetching into the log.

While the eFetch persistent write tracking reduces performance and energy degradation by ensuring that traditional prefetching policies are aware of log updates, eFetch performs one step further to leverage prefetching to improve the performance and energy of working data

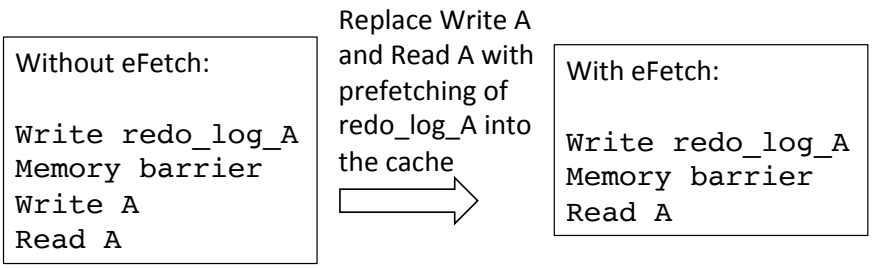
access. The described write-tracking and address-tracking opens up for the next two mechanisms to work and improve persistent memory system performance.

2.3.2 Redo Log Prefetching

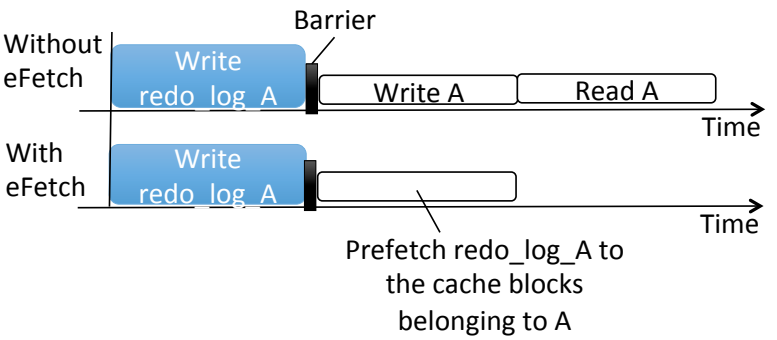
The key idea of this mechanism is to perform working data updates in cache and subsequent reads of the same data simultaneously with a single redo log prefetching (Figure 2.3). Efetch does not cache the redo log. Therefore, as shown in the figure, the CPU is supposed to issue new write requests to the working data **A**. Depending on the size of data **A**, the write can consists of one or several store instructions. Subsequent reads of **A** need to wait until `Write A` is performed. It is fine if the size of the data is small. But persistent memory systems can manipulate large data structures in databases and file systems, where waiting for `Write A` and then performing `Read A` take a long time. Instead of doing so, eFetch allows `Read A` to start right after `Write redo_log_A` is completed; `Read A` will trigger the prefetching of redo_log_A from NV memory into the cache blocks that are supposed to store the working data **A**. As such, eFetch leverages the redo log to update the working data without relying on the CPU to issue new requests; meanwhile, the subsequent read requests are serviced.

Accessing Data from Log. In most logging schemes, log updates rely on key-value information for referencing updates done or to be done to the persistent data structure. To maintain such information, the redo log records the new value of that data element (to be "redone") and the location of the actual persistent data item within the working data updated. These two values are coupled within a single log entry. The latter allows obtaining the address of the value of the working data and cache that data block instead of the block containing the log. This also allows us to use the address to trigger eFetch to prefetch additional members of the data into the cache. Doing so increases hits in the cache by having the data values and adjacent addresses already in the cache on future accesses. It also improves the performance of further log requests and updates, which rely on the same scheme during execution. As shown in Figure 2.3, in the case of a cache hit on the cache block containing **A** during the redo log update of **A**, no additional work is required aside from updating the value of **A** within that block.

Shrinking Persistent Memory Footprint in Cache. There is an additional advantage to accessing data directly during the log update and caching that instead. A log update requires a key-value pair (two values) to store in memory. A regular persistent data item only requires



(a) Example codes.



(b) Timeline.

Figure 2.3: Redo log prefetching mechanism. (a) Example application (or system software) codes without and with the proposed redo log prefetching mechanism. (b) A timeline that illustrates the saved time by using this mechanism.

one value to store in memory and therefore also in the cache. By replacing memory requests to the log with those of the desired persistent structure, this design reduces the memory footprint of persistent memory in the cache. The original log request would store one value instead of two values, thus halving the memory footprint. Furthermore, accommodating a separate cache block to be used for the persistent data later after the log request is unnecessary. Instead, eFetch processes both the log request and the persistent memory access together, saving time as well as space in the cache to be used for accommodating more reusable data.

2.3.3 Undo Logging Triggered Prefetching

The idea of this mechanism is to proactively start working data updates, so that they complete earlier than usual in undo-log-based persistent memory. As shown in Figure 2.4, the

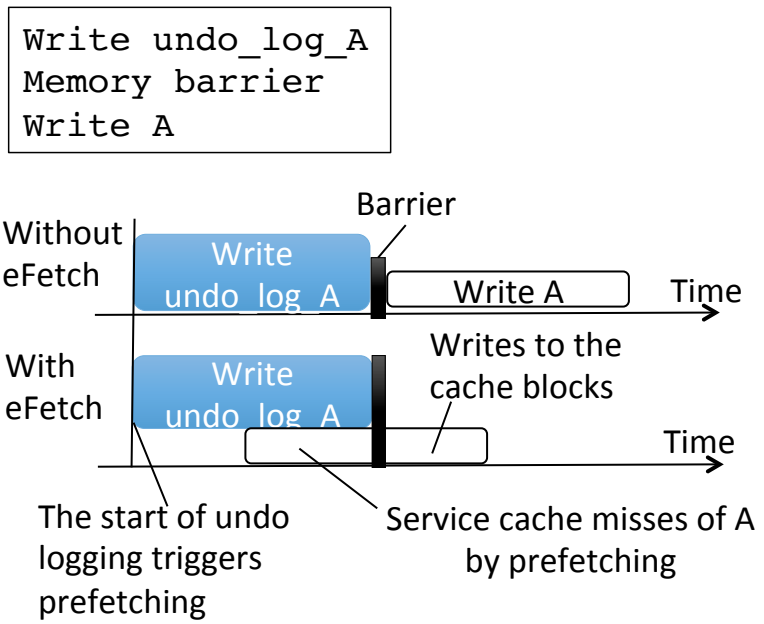


Figure 2.4: Undo logging triggered prefetching

level-1 (L1) cache needs to fetch data **A** from lower levels, if `Write A` misses in the cache; an working data update (`Write A`) therefore involves two steps: 1) servicing cache misses and 2) updating the cache blocks corresponding to **A**. EFetch proactively performs the first step by leveraging the observation that working data updates will follow undo log updates in WAL. In addition, prefetching would only load the data from lower levels in the memory hierarchy; it does not modify data values and therefore does not violate data persistence. EFetch leverages such observations to proactively trigger prefetching of working data. As illustrated in Figure 2.4, the eFetch implementation ensures that subsequent writes to the fetched cache blocks corresponding to **A** are performed after `Write undo_log_A` is written into NV memory.

Distinguishing Undo Log from Redo Log. From the perspective of the prefetcher in hardware, there is no difference between an application that uses redo log and one that uses undo log. Both are uncacheable and require a reserved address space in which no prefetching can occur. However, the key difference is how many writes occur in hardware and whether the software still requires to execute write instructions update the working data. Efetch distinguishes between undo and redo log in its software interface. This is further discussed in Section 2.4. Unlike in the redo log case, the write cannot directly perform be performed to the working data (the main

persistent data structure), because the new data values are not yet available. Therefore, with undo logging, eFetch still requires the write_A in the code to properly update the values.

Preemptive Cache Write-Allocate. Similar to the redo log, the undo log mechanism makes use of the information packed with the update of the undo log request to expedite following requests to the persistent data structure. The difference with processing undo logs is that it is not yet known what the new value to write to the working data will be, only the old value (to be "undone") is known. However, the undo log access is treated like the main data access. Like with redo logging, an undo log update provides the address of where the element in the working data will be placed. This address is used to preemptively fetch the cache block containing the working data from memory.

Normally in a cache, all write request misses must be write-allocated regardless, this is a standard property of most caches [108]. This cache block allocation typically occurs during cache misses on writes. If the subsequent write to the original data ends up missing in the cache, the cache will have to perform an allocate regardless. EFetch can start to perform the cache write-allocate process early. This leads to a significant reduction in the chance of a cache miss on the write to the working data, because the cache will already be allocated. This expedites the working data write requests and completes them sooner than usual. In the case of a cache hit during the initial log update cache request, no additional work is needed because the cache block containing the working data and its old value (the undo value) should already be in the cache. However, since eFetch treat this as an access to the working data, it will also prefetch additional address blocks related to this data as part of its prefetching algorithm.

2.3.4 Putting It All Together

Figure 2.1 illustrate a step-by-step overview of how all three mechanisms discussed in this section work together to implement eFetch.

1. A persistent memory update occurs in the code. This is denoted by the use of a redo or undo log call followed by a memory fence. This can be detected at compile time.

2. At all times during execution of the code, eFetch is running its prefetching policy (e.g., next-line, stride, locality-based, Markov, etc.). EFetch uses a reservation buffer check to determine which generated prefetch requests are for addresses in memory containing a the log. These requests are ignored.

3. There is an undo or redo log update in the code.
    (a) If it is a redo log, code does not require an update to the persistent structure.
    (b) If it is an undo log, code looks similar to implementation without eFetch.
4. The undo or redo log update occurs.
    (a) In either case, log update bypasses the cache and goes straight to NVRAM. This is triggered by the address of the store instructions lying within the address space of a log data structure.
    (b) While requesting, eFetch detects the request belongs to a log and instead generates a cacheable request for the persistent member belonging to that log. For both forms of logging, eFetch obtains the address of this member from the provided entry.
    (c) If this new cacheable request hits in the undo log case, request and log update end and no further work is required.
    (d) If this new cacheable request hits in the redo log case, the cache block containing the persistent data value is updated with the new value, provided in the log entry.
    (e) If this new cacheable request misses in the cache in either log case, the processor must allocate the block in the cache as per write-allocate policy.
    (f) After write allocation, the fetch request will also write to that new cache block the new value provided with the log update if it is a redo log.
    (g) eFetch generates additional memory requests according to its algorithm.
5. The write request to the persistent data structure occurs after the memory fence but only in the case of an undo log. It is not necessary in the redo case because the new value is already in the cache in the correct block.

2.4 Implementing eFetch

This section addresses implementation details of eFetch, including software interface, hardware prefetcher design (log address buffer, prefetching policies, and prefetching controller), and hardware memory barrier.

Software Interface. eFetch software interface provides software programmers or runtime libraries (can be used in either case) with a way to define log and working data updates, and dependent reads. These are software hints that eFetch hardware takes to determine the appropriate prefetching mechanisms. For example, eFetch uses a pair of log functions, `redo_log()` and `undo_log()`, which defines redo and undo log updates, respectively. In redo-log-based applications, software can leverage the *redo log prefetching* mechanism by eliminating those working data updates followed by reads of the same data. An example is shown in Figure 2.3(a) (the example code with redo log prefetching). EFetch also provides software with an interface, `log_alloc()` and `log_dealloc()`, to allocate and deallocate redo/undo log entries.

Log Address Buffer. eFetch employs a log address buffer in the hardware prefetcher. Persistent memory applications can allocate one or several consecutive memory regions to store the log. Furthermore, the log size can grow during application execution, creating new memory regions for log or growing size of a memory region. With its software interface, eFetch can receive corresponding hints on the change of the memory regions from software and adjusts the address buffering accordingly. The evaluation adopts a ten-entry buffer, which can accommodate the evaluated applications very well.

Figure 2.5 shows an example of a log address buffer with address checking logic that produces a valid signal determining whether or not the prefetching request is even valid. The logic shows an exhaustive check for the addresses, which is similar to a cache tag array of a fully-associative cache. Yet full-associativity is not very pragmatic for most cache designs. Ideally, the address check in the prefetcher needs to use the same associativity as the cache itself, because it is possible for the same tag to exist across multiple cache lines without there being a conflict miss. However, the hardware prefetcher does not have knowledge of the cache structure, so treating it as a fully-associative helps account for the worst-case scenario. However, if information about the cache's structure was available to the prefetcher, this log address buffer could match its associativity when performing the address check. This would lead to even better performance than the evaluation shows.

Prefetching Policies. eFetch can employ various prefetching policies, such as next-line, stride, locality-based, and Markov prefetching, with ordinary (not log updates, working data updates, or dependent reads) data accesses. The *persistent write tracking* only ensures that existing prefetching policies bypass the prefetching of memory regions for log. On top of the prefetch-

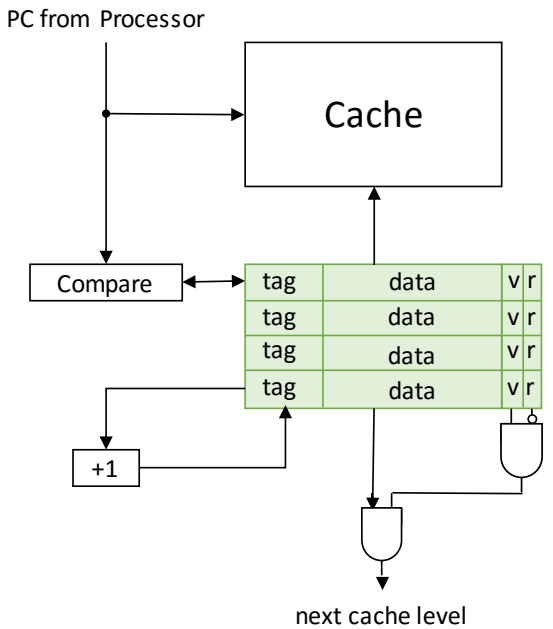


Figure 2.5: An example of a log address buffer with address checking logic that produces a valid signal determining whether or not the prefetching request is even valid.

ing policy used (which can be any of the various policies), eFetch performs *redo log prefetching* to manipulate log updates, working data updates, and dependent reads with redo-log-based persistent memory applications. If a persistent memory application employs undo logging instead, eFetch performs *undo logging triggered prefetching* on top of the prefetching policies used.

Prefetching Controller. The eFetch prefetching controller performs log address tracking; it also initiates prefetching requests based on the prefetching policies used and determines the proposed prefetching mechanisms. Yet NV memory bandwidth is a critical resource for persistent memory application performance. Prefetching can hurt performance, if it interference with ordinary memory requests. Therefore, the eFetch prefetching controller monitors the NV memory bandwidth utilization, which is calculated as the ratio between the demanded memory bandwidth and peak memory bandwidth. The eFetch prefetching controller calculates demanded memory bandwidth by periodically sampling the hardware counter events such as NV memory reads and writes within a given time period. The peak memory bandwidth is known as memory bus frequency $\times$ bus width.

Hardware Memory Barrier. The eFetch *Undo logging triggered prefetching* proactively ser-

vices L1 cache misses of working data updates before completing undo logging updates. However, to ensure data persistence, writes to these prefetched cache blocks cannot be performed until after the undo logging updates are written into NV memory. As such, eFetch needs to adopt a hardware memory barrier between undo log updates and writes to the cache blocks of the working data. The hardware memory barrier is implemented in L1 cache controller. The controller will reject the writes to the prefetched cache blocks until it receives a signal from the memory controller, indicating that the log updates are already written into the NV memory. Note that low-level persistent memory instructions, such as Intel's `pcommit` can ensure that the log writes arrive at NV memory by forcing them out of processor buffers.

2.5 Prefetching and Persistent Memory Experiments

This section describes the experiments and the tools needed to perform evaluation on eFetch.

2.5.1 Framework

This chapter uses McSimA+ [11] to simulate the systems described in the previous Design section. This simulator, although it has fewer configuration *options* than alternate simulators, provides a simpler configuration interface. The source code is modular and flexible for modifying. It allows iterating more quickly on design ideas. Additionally, there is prior persistent memory research done with this simulator [148]. This reassures the confidence of the results.

The experiments configure the simulator to use a multi-core out-of-order processor, shown in Table 2.1. Each L1 cache has a 64-entry stride prefetcher. This means that its *depth* and *stride* are 64. Note there is no L3 cache in this simulator, but the L2 is configured to match the Last Level Cache (LLC) of the corresponding processor. The prefetcher's stride policy is not required for eFetch. Any prefetching policies may be used with eFetch's mechanisms. Efetch uses stride because it is commonly used and can speedup the chosen persistent benchmarks. This is further discussed below and the effects of a standard stride prefetcher is shown in Section 2.6.

Processor	Similar to Intel Core i7 / 22 nm
Cores	4 cores, 2.5GHz, 2 threads/core
IL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency, 64-entry stride prefetcher
DL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency, 64-entry stride prefetcher
L2 Cache	8MB, 16-way set-associative, 64B cache lines, 4.4ns latency
Memory Controller	64-/64-entry read/write queues
NVRAM DIMM	8GB, 8 banks, 2KB row 36ns row-buffer hit, 100/300ns read/write row-buffer conflict [76].
Power and Energy	Processor: 149W (peak) NVRAM: row buffer read (write): 0.93 (1.02) pJ/bit, array read (write): 2.47 (16.82) pJ/bit [76]

Table 2.1: Processor and memory configurations.

2.5.2 Benchmarks

This section evaluates common persistent memory microbenchmarks in the experiments. The microbenchmarks randomly update persistent data structures, performing functions such as value swaps, key-value stores, and item deletions. The data structures used are: hash table, red-black tree, array, B+tree, and graph. These structures are studied in previous works on persistent memory [30].

Table 2.2 summarizes the these microbenchmarks. The experiments later in this chapter use multiple versions of each benchmark, varying data types and data set sizes. Persistent structures with integer elements cache less data per write, whereas their string version require many cache blocks per update. This shows the affect of eFetch on applications with both complex and lightweight data structures. These benchmarks are compiled in native x86 to run on McSimA+ through Intel's pin tool Pthreads.

First, this thesis runs a sensitivity study on these microbenchmarks. The sensitivity study compares the results and performance of these benchmarks without any prefetching to versions with a standard prefetcher (not eFetch). These two experiments are run without

Name	Memory Footprint	Description
Hash [30]	256 MB	Searches for a value in an open-chain hash table. Insert if absent, remove if found.
RBTree [146]	256 MB	Searches for a value in a red-black tree. Insert if absent, remove if found
SPS [146]	1 GB	Random swaps between entries in a 1 GB vector of values.
BTree [22]	256 MB	Searches for a value in a B+ tree. Insert if absent, remove if found
SSCA2 [17]	16 MB	A transactional implementation of SSCA 2.2, performing several analyses of large, scale-free graph.

Table 2.2: A list of evaluated microbenchmarks.

persistent memory. This allows comparing the differences in adding prefetching between persistent and non-persistent systems. The results are meant to motivate eFetch and address issues discussed in Section 2.2.

The primary experiment of this chapter implements eFetch in the persistent applications and compares it to a baseline system with a standard prefetcher. These experiments are run in single-core, 2-core, and 4-core system configurations. The sensitivity study only runs single-core because it's used primarily for motivation. However, this thesis wants to analyze eFetch's scalability across multi-core configurations. This is important for persistent memory applications because it's better representative of real-world use cases in database systems and high-performance computing.

2.6 eFetch Evaluation

To demonstrate the advantages of eFetch and motivate its need, the experiments study the following metrics: instructions per cycle (IPC), operational throughput, dynamic memory energy consumption, and overall memory traffic. Note that The experiments look at overall memory traffic, not just write traffic. Persistent memory research typically focuses on writes because they are significantly more expensive than reads. However, prefetching directly impacts read request so reads are included in the memory traffic. Operational throughput refers to

the the number of executed application-level operations per unit time. The operations vary with benchmarks, see Section 2.5. In the microbenchmarks, the experiments scale number of operations with the size of the data structure, reflected under "memory footprint" in Table 2.2. However, in the string versions of the *hash*, *rbtree*, and *sps* benchmarks scale up the size per element rather than the number of operations.

All numerical values shown are speedup values (higher is better). This speedup is relative to a baseline implementation. The primary baseline used is a persistent memory benchmark with a standard prefetcher (not eFetch). Note that in the case of dynamic memory energy and memory traffic, this "speedup" refers to a reduction factor. Therefore, all speedup values over 1 for all metrics corresponds to an improvement. The dashed lines represents the reference baseline for that particular experiment.

Performance Sensitivity to Prefetching in Persistent Memory. To motivate (Section 2.2) eFetch (Section 2.3) and experiments (Section 2.5), this thesis runs a sensitivity study showing the average effects of prefetching on persistent memory applications across all benchmarks and metrics. Figure 2.6 shows this impact. The left bar is the baseline case showing the average speedup of prefetching for non-persistent versions of the benchmarks. This value is treated as a "target" because eFetch should be just as good, if not better. The right bar is the average speedup of the baseline prefetching (Section 2.5) in the persistent memory benchmarks. Note that prefetching does not necessarily improve but hurts performance in some benchmarks. This demonstrates the fact that not all prefetch policies benefit all applications equally (further discussed in Section 2.2). It also motivates the research and the need for eFetch.

The main experiments expand on this sensitivity study, showing the details of each microbenchmark individually. For each benchmark, multi-core versions of each are run. The experiments study single-core, duo-core, and quad-core implementations. This allows observing the scalability of eFetch. Each core has two L1 caches, and each cache has a prefetcher. The data caches have eFetch while the instruction caches use a standard stride prefetcher. This thesis makes the following major observations out of the evaluation results.

Performance Analysis. Overall there is an improvement in IPC (Figure 2.7), operational throughput (Figure 2.8), memory energy (Figure 2.9), and memory traffic (Figure 2.10) with the proposed eFetch mechanisms. The eFetch design improves IPC by up to 1.7× over the baseline, with a 1.33× average speedup across all benchmarks. Operation throughput increases

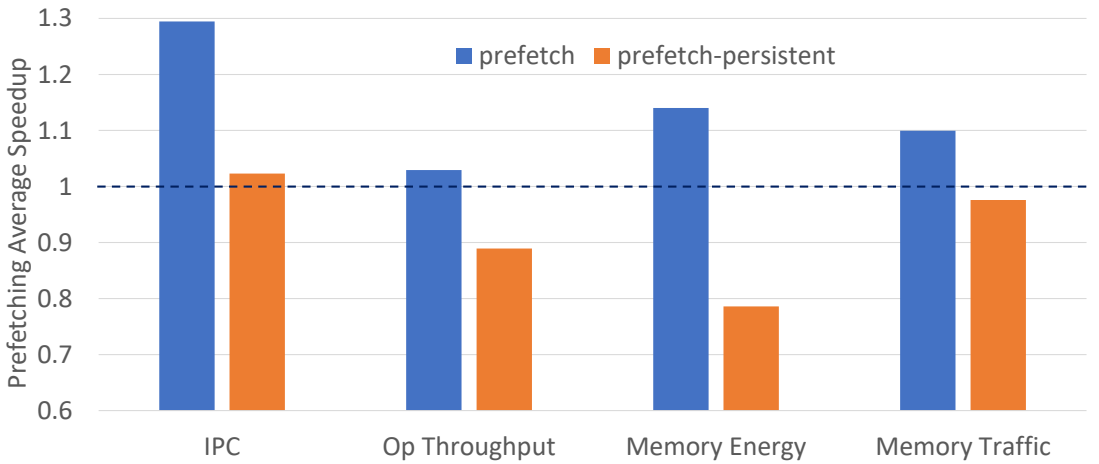


Figure 2.6: Sensitivity study showing the average speedup across all benchmarks between persistent and non-persistent applications. The over-prefetching of log data by the prefetcher in persistent memory applications is detrimental to its performance.

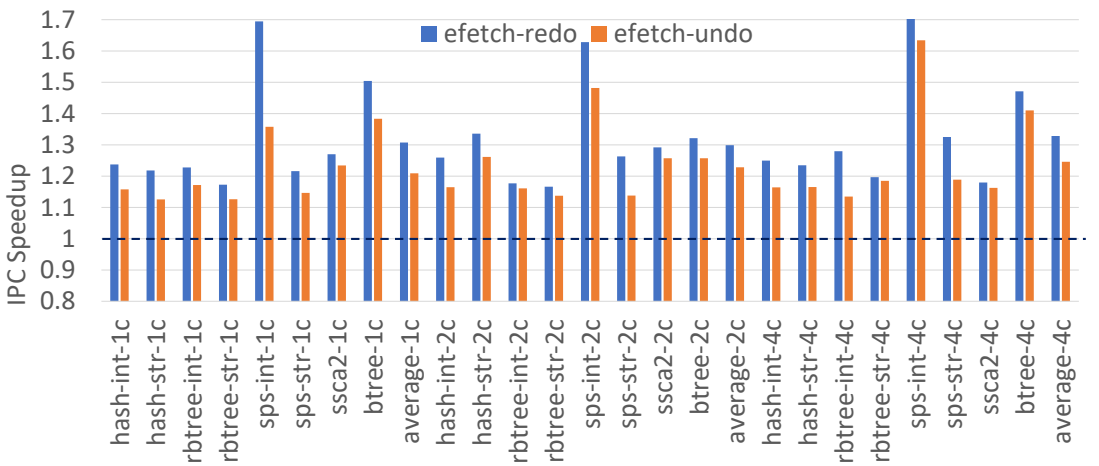


Figure 2.7: IPC speedup of microbenchmarks (normalized to the unsafe baseline, dashed line)

by up to 1.56×, with average of 1.24× speedup. eFetch reduces NVRAM memory traffic, by up to 1.79× and an average factor of 1.36×. Related to memory traffic reduction, eFetch decreases dynamic memory energy consumption by up to 1.82× and an average factor of 1.39× across all benchmarks. The overall decrease in memory accesses leads to a decrease in the Average Memory Access Latency (AMAL) which directly improves IPC and thus operational throughput.

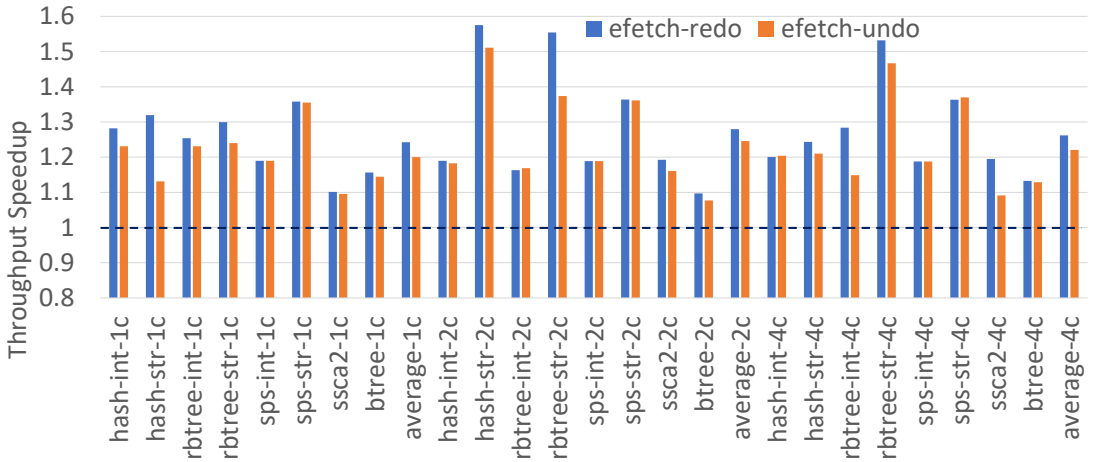


Figure 2.8: Operational throughput speedup of microbenchmarks (normalized to the unsafe baseline, dashed line).

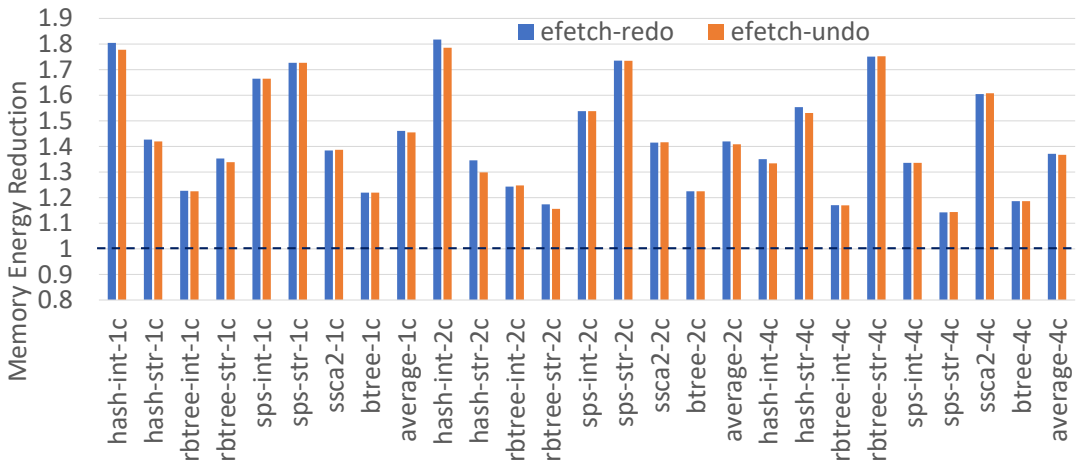


Figure 2.9: Dynamic energy reduction across all microbenchmarks (normalized to unsafe baseline, dashed line).

Multicore Scalability. Results show that there are not significant differences between benchmarks with a different number of cores. Visual or numerical differences between corresponding benchmarks (e.g., *hash-int-1c* and *hash-int-2c*) are all within a small error margin. This error margin is caused by different random seeds between benchmarks, whose effects propagate through multi-core systems differently depending on the number of cores. Note also, however, that there is not a clear pattern to the results from increasing or decreasing the number of cores.

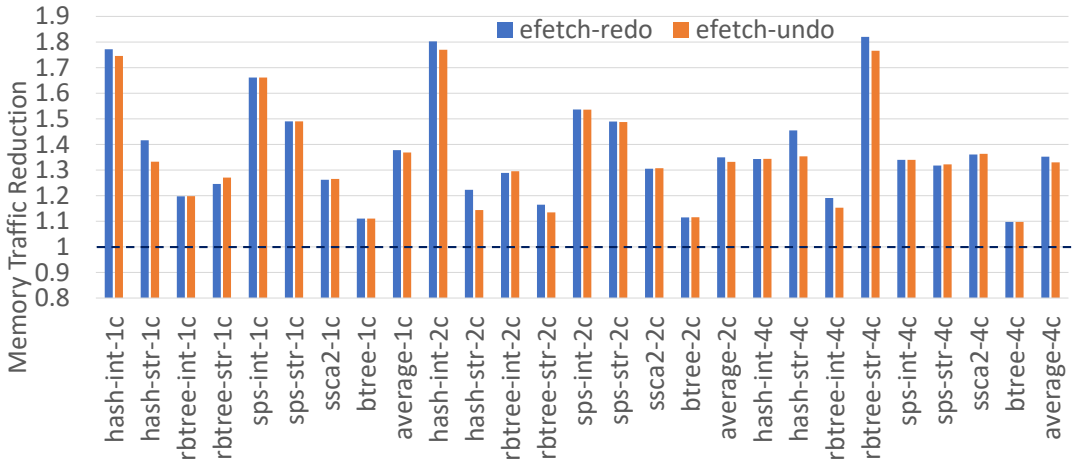


Figure 2.10: Memory write traffic reduction across all microbenchmarks (normalized to unsafe baseline, dashed line).

This is shown by *average-*c* results which are all within 5% of each other.

Redo vs Undo Logging. Note that there is a difference in performance between redo and undo logging in each result. This difference stems from the difference between the two mechanisms (more details in Section 2.3). In the best case scenario, undo logging achieves as good a speedup as redo logging. In most cases, however, undo logging performs worse than redo logging, but not worse than the baseline. Benchmarks with redo and undo logging are most similar with regard to memory traffic and memory energy consumption. The reason is that the mechanisms do not add or remove total accesses to memory during prefetching or on the initial log update. There are discrepancies between redo and undo logging under eFetch in the results of IPC and operational throughput. Recall that the undo logging mechanism still requires the instruction call to update the data value. The eFetch redo logging mechanism always removes this need, leading to fewer instructions through the pipeline and thus a better IPC and throughput. Both redo and undo logging reap the benefits of eFetch, but persistent memory applications with redo logging benefits more.

Performance Recovery and eFetch Summary. Figure 2.11 is similar to Figure 2.6 in structure. The leftmost two bars of each cluster correspond to the same bars in the sensitivity study. This graph shows the results of eFetch in comparison to the baseline without eFetch. Recall that the goal of eFetch was to recover the performance loss from prefetching with persistent memory. Figure 2.11 shows that eFetch either meets the "target" (i.e., the leftmost bar) or surpasses

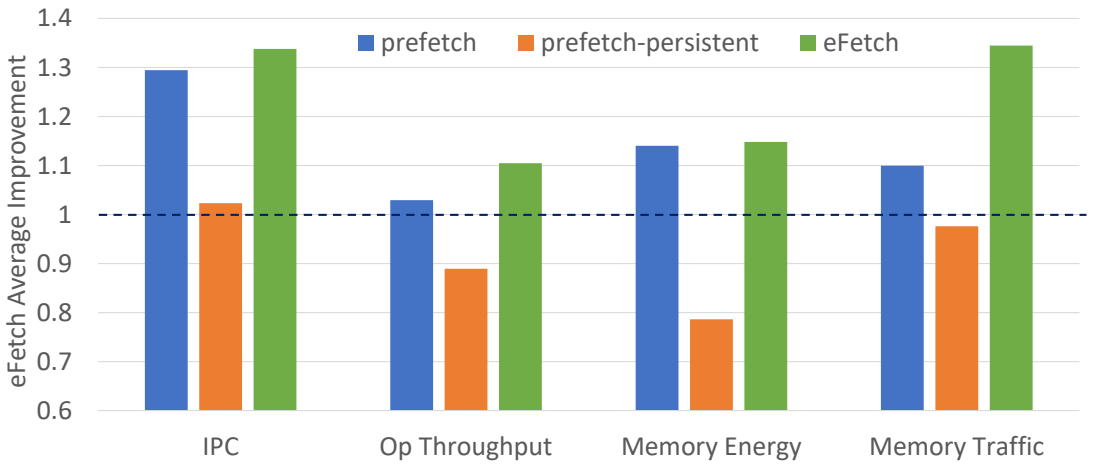


Figure 2.11: Shows the average improvement of eFetch across all benchmarks, shown in comparison with sensitivity study.

it entirely. This shows that eFetch is viable and compares not only with baseline persistent memory implementation with standard prefetching, but also with prefetching for non-persistent applications.

2.7 eFetch Hardware Overhead

Impact on Instruction Count and Cycle Time. Two metrics the evaluation does not explore is the program instruction count and the cycle time.

For program instruction count, eFetch does not change the instruction count of persistent memory programs that use undo log. Section 2.3 discusses this further. However, persistent applications that use redo logging does reduce the instruction count. Every write to persistent data objects typically requires at least two writes (one for the log, one for the data). By removing one of the writes, eFetch can halve the total memory write traffic (and thus store instructions) in the best case scenario. In the worst case scenario, one store instruction is removed per transaction.

A problem with hardware-based solutions arises when the physical cost of the design outweighs the benefit. This occurs when the design is too complex that it explodes metrics like chip area, power consumption, or cycle time. The eFetch implementation and experiments use a functional level simulator, too high-level to give a good analysis of how much

Mechanism	Logic Type	Size
Prefetch reserve bits	flip-flops	128 Bytes
Log head pointer register	flip-flops	8 Bytes
Log tail pointer register	flip-flops	8 Bytes
eFetch Muxes/Decoders	medium scale integration	
eFetch Logic Gates	small scale integration	

Table 2.3: Summary of major hardware overhead.

area or power overhead eFetch has. For cycle time, however, the eFetch changes to hardware prefetching are not on the critical path of the processor. This is consistent with similar works on prefetching [126] [48]. Therefore, eFetch should not slow clock frequency from the additional hardware.

Implementation Overhead Estimation. Yet, this thesis can make an estimation for eFetch's hardware overhead. Table 2.3 shows this overhead. The few components added to the hardware outline the simplicity of the design. Note that the size of the overhead is determined by the size of the prefetcher. In the eFetch implementation and experiments, eFetch is set to be large and aggressive enough to prefetch up to 64 addresses' worth of data. The amount of overhead is also determined by the number of prefetchers in the overall system. The implementation in this chapter has one prefetcher per L1 cache. The aggressiveness and prefetch policy may be tuned down to save on hardware overhead.

Memory Consistency and Transaction Model. The eFetch design and experiments use the same memory consistency model as an i7 processor. Section 2.5 outlines further details of the system. The impact of eFetch has no implication on either the consistency or transaction model. It's possible to use eFetch on legacy code, but will not yield the full advantages without changing their software model to match the eFetch model. EFetch does not require strict ordering criteria in its benchmarks like other models.

In other software-based logging schemes like [16], their ordering of logging and updating persistent structures is flipped. Their write-behind logging design performs the update first, and this appears to violate the eFetch modeling expectations. However, this simply means that implementing eFetch on their model will not incur benefits from the redo and undo logging mechanisms. However, it will still benefit from the eFetch write-tracking mechanism and removal of over-prefetching into the cache of log data.

2.8 Discussion on Uncacheable Logging versus Cacheable Logging

One major question this thesis had when designing eFetch was about whether the log should be cacheable or uncacheable. As noted earlier in the chapter, this thesis assumes an uncacheable log for the persistent memory system which is consistent with previous studies. However, this assumption is not a given and in designing the hardware and simulator for eFetch, the implementation of uncacheable logging raised questions about how well supported it is in x86, the ISA of the simulator. In theory, caching the log should contaminate critical cache resources, thus degrading performance.

This motivated contribution in a previous study analyzing the differences and trade-offs between cacheable and uncacheable logging [79]. That study showed that cache bypassing in commodity processors (e.g. x86) is sub-optimal for supporting uncacheable log writes. Specifically, bypassing the caches for logging is shown to degrade persistent memory system performance compared to cacheable logging. In particular, common processors use special instructions and a write-combining buffer to implement uncacheable writes. Those instructions end up requiring more cycles and does not perform as well as experiments with cached writes.

A big factor that contributes to this performance loss is due to the write-combining buffer and how these uncacheable writes coalesce before writing out to memory. In current implementations of uncacheable logging, this write-combining buffer writes the log at a granularity smaller than a cache line, causing partial writes. This means there is an underutilization of hardware resources and hence lower performance. That paper still notes, however, that there are still observed advantages to uncacheable logging that comply with the theory (e.g., lower miss rates in the cache). This suggests that the problem lies in the implementation rather than the concept. It also proposes that a hardware-based approach for persistent memory would indeed benefit from a more holistic approach that covers multiple parts of the microarchitecture. The next chapter in this thesis takes this holistic approach and corrects these shortcomings with uncacheable logging. This will further strengthen the case for hardware and architecture solutions for persistent memory.

Chapter 3

Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems

Persistent memory is a new tier of memory that combines the benefits of both storage systems and main memory. It has the data persistence of storage with the fast load/store interface of memory. Most previous persistent memory designs place careful control over the order of writes arriving at persistent memory. This can prevent caches and memory controllers from optimizing system performance through write coalescing and reordering. This thesis identifies that such write-order control can be relaxed by employing undo+redo logging for data in persistent memory systems. However, traditional software logging mechanisms are expensive to adopt in persistent memory due to performance and energy overheads. Previously proposed hardware logging schemes are inefficient and do not fully address the issues in software.

To address these challenges, this thesis proposes a hardware undo+redo logging design which maintains data persistence by leveraging the write-back, write-allocate policies used in commodity caches. Furthermore, this thesis develops a cache force-write-back mechanism in hardware to significantly reduce the performance and energy overheads from forcing data into persistent memory. The evaluation across persistent memory microbenchmarks and real workloads demonstrates that the design described in this thesis significantly improves system throughput and reduces both dynamic energy and memory traffic. It also provides strong con-

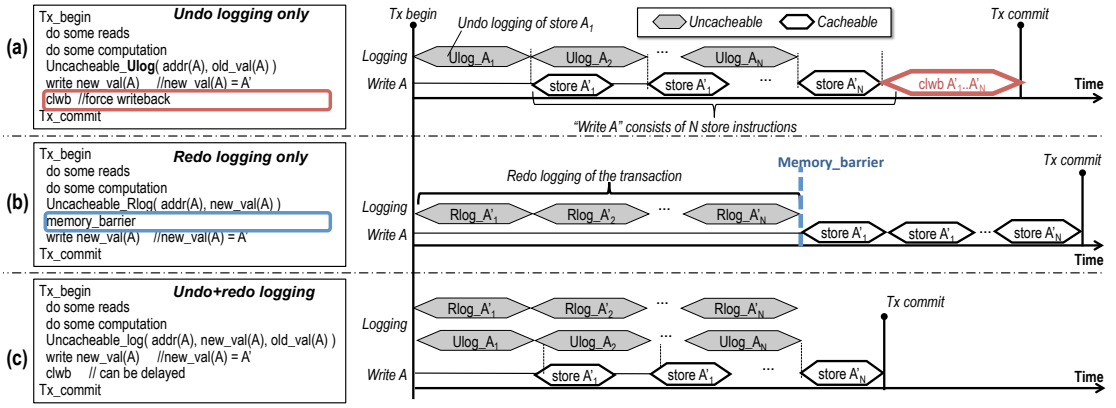


Figure 3.1: Comparison of executing a transaction in persistent memory with (a) undo logging, (b) redo logging, and (c) both undo and redo logging.

sistency guarantees compared to software approaches. This chapter elaborates on the design of hardware undo+redo logging.

3.1 Background on Persistent Memory and Logging

Persistent memory is fundamentally different from traditional DRAM main memory or their NVRAM replacement, due to its persistence (i.e., crash consistency) property inherited from storage systems. Persistent memory needs to ensure the integrity of in-memory data despite system crashes and power loss [10, 25, 64, 72, 85, 88, 89, 98, 109, 111, 135]. The persistence property is not guaranteed by memory consistency in traditional memory systems. Memory consistency ensures a consistent global view of processor caches and main memory, while persistent memory needs to ensure that the data in the NVRAM main memory is standalone consistent [72, 109, 135].

3.1.1 Persistent Memory Write-order Control

To maintain data persistence, most persistent memory designs employ transactions to update persistent data and carefully control the order of writes arriving in NVRAM [16, 109, 135]. A transaction (e.g., the code example in Figure 3.1) consists of a group of persistent memory updates performed in the manner of "all or nothing" in the face of system failures. Persistent memory systems also force cache write-backs (e.g., `clflush`, `clwb`, and `dccvap`)

and use memory barrier instructions (e.g., `mfence` and `sfence`) throughout transactions to enforce write-order control [16, 30, 73, 135, 146].

Recent works strived to improve persistent memory performance towards a native non-persistent system [73, 109, 146]. In general, whether employing logging in persistent memory or not, most face similar problems. (i) They introduce nontrivial hardware overhead (e.g., by integrating NVRAM cache/buffers or substantial extra bookkeeping components in the processor) [116, 146]. (ii) They fall back to low-performance modes once the bookkeeping components or the NVRAM cache/buffer are saturated [73, 146]. (iii) They inhibit caches from coalescing and reordering persistent data writes [146] (details discussed in Section 5.2).

Forced cache write-backs ensure that cached data updates made by completed (i.e., committed) transactions are written to NVRAM. This ensures NVRAM is in a persistent state with the latest data updates. Memory barriers stall subsequent data updates until the previous updates by the transaction complete. However, this write-order control prevents caches from optimizing system performance via coalescing and reordering writes. The forced cache write-backs and memory barriers can also block or interfere with subsequent read and write requests that share the memory bus. This happens regardless of whether these requests are independent from the persistent data access or not [85, 148].

3.1.2 Why Undo+Redo Logging

While prior persistent memory designs only employ either undo or redo logging to maintain data persistence, this thesis observes that using both can substantially relax the aforementioned write-order control placed on caches.

Logging in persistent memory. Logging is widely used in persistent memory designs [30, 72, 73, 135]. In addition to working data updates, persistent memory systems can maintain copies of the changes in the log. Previous designs typically employ either undo or redo logging. Figure 3.1(a) shows that an undo log records old versions of data before the transaction changes the value. If the system fails during an active transaction, the system can roll back to the state before the transaction by replaying the undo log. Figure 3.1(b) illustrates an example of a persistent transaction that uses redo logging. The redo log records new versions of data. After system failures, replaying the redo log recovers the persistent data with the latest changes tracked by the redo log. In persistent memory systems, logs are typically *uncacheable* because they are

meant to be accessed only during the recovery. Thus, they are not reused during application execution. They must also arrive in NVRAM in order, which is guaranteed through bypassing the caches.

Benefits of undo+redo logging. Combining undo and redo logging (undo+redo) is widely used in disk-based database management systems (DBMSs) [95]. Yet, this thesis finds that persistent memory design can leverage this concept to relax the write-order constraints on the caches.

Figure 3.1(a) shows that uncacheable, store-granular undo logging can eliminate the memory barrier between the log and working data writes. As long as the log entry ($Ulog\_A_1$) is written into NVRAM before its corresponding store to the working data ($store\ A'_1$), the partially completed store can be undone after a system failure. Furthermore, $store\ A'_1$ must traverse the cache hierarchy. The uncacheable $Ulog\_A_1$ may be buffered (e.g., in a four to six cache-line sized entry write-combining buffer in x86 processors). However, it still requires much less time to get out of the processor than cached stores. This naturally maintains the write ordering without explicit memory barrier instructions between the log and the persistent data writes. That is, logging and working data writes are performed in a pipeline-like manner (like in the timeline in Figure 3.1(a)). However, a downside is that undo logging requires a forced cache write-back before the transaction commits. This is necessary to recover the latest transaction state after system failures. Otherwise, the data changes made by the transaction will not be committed to memory.

Instead, redo logging allows transactions to commit without explicit cache write-backs because the redo log, once updates complete, already has the latest version of the transactions (Figure 3.1(b)). However, memory barriers are necessary to complete the redo log of A before any stores of A reach NVRAM. The dashed blue line in the timeline shows this ordering constraint. Otherwise, a system crash when the redo logging is incomplete, while working data A is partially overwritten in NVRAM (by $store\ A'_k$), causes data corruption.

Figure 3.1(c) shows that undo+redo logging combines the benefits of both. This can eliminate the memory barrier, as a result, between the log and persistent writes. A forced cache write-back (e.g., `clwb`) is unnecessary for an unlimited sized log. However, it can be postponed until after the transaction commits for a limited sized log (Section 3.1.3).

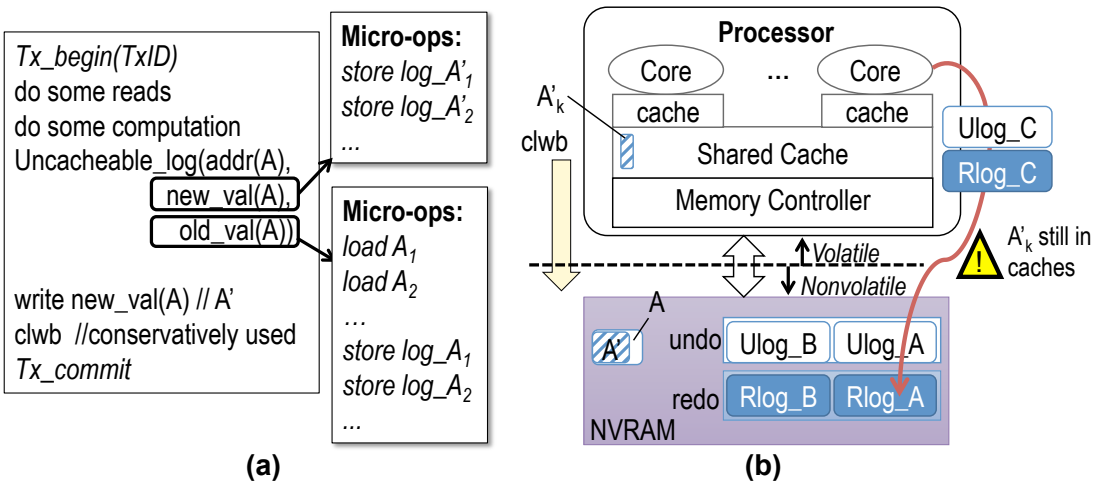


Figure 3.2: Inefficiency of logging in software.

3.1.3 Why Undo+Redo Logging in Hardware

Though promising, undo+redo logging is not used in persistent memory system designs because previous software logging schemes are inefficient (Figure 3.2).

Extra instructions in the CPU pipeline. Logging in software uses logging functions in transactions. Figure 3.2(a) shows that both undo and redo logging can introduce a large number of instructions into the CPU pipeline. As demonstrated in the experimental results (Section 3.5), using only undo logging can lead to more than doubled instructions compared to memory systems without persistent memory. Undo+redo logging can introduce a prohibitively large number of instructions to the CPU pipeline, occupying compute resources needed for data movement.

Increased NVRAM traffic. Most instructions for logging are loads and stores. As a result, logging substantially increases memory traffic. In particular, undo logging must not only store to the log, but it must also first read the old values of the working data from the cache and memory hierarchy. This further increases memory traffic.

Conservative cache forced write-back. Logs can have a limited size[1]. Suppose that, without losing generality, a log can hold undo+redo records of two transactions (Figure 3.2(b)). To log a third transaction ($Ulog\_C$ and $Rlog\_C$), an existing log record must be overwritten, say $Ulog\_A$ and $Rlog\_A$ (transaction A). If any updates of transaction A (e.g., A'_k) are still in caches, these

[1] Although log size can grow on demand, this introduces extra system overhead on managing variable size logs [135]. Therefore, this thesis studies fixed size logs.

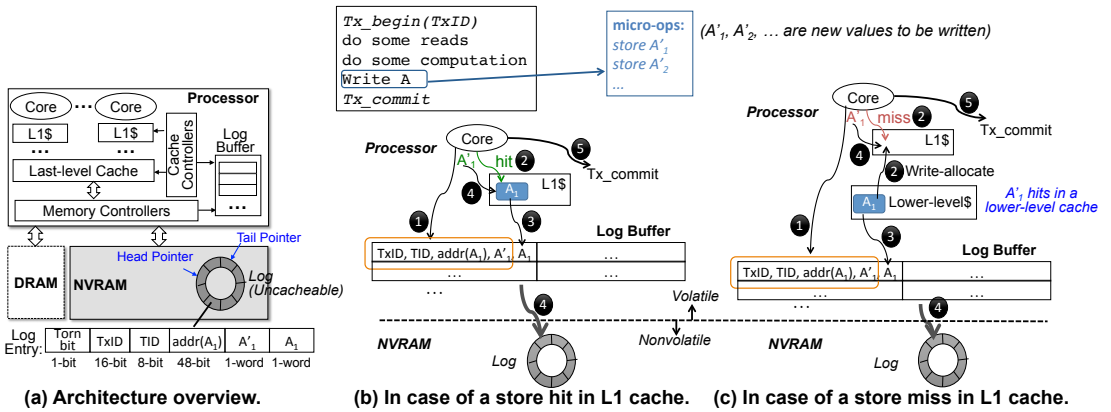


Figure 3.3: Overview of the proposed hardware logging in persistent memory.

updates must be forced into the NVRAM before overwriting their log entry. The problem is that caches are invisible to software. Therefore, software does not know whether or which particular updates to A are still in the caches. Thus, once a log becomes full (after garbage collection), software may conservatively force cache write-backs before committing the transaction. This unfortunately negates the benefit of redo logging.

Risks of data persistence in multithreading. In addition to the above challenges, multithreading further complicates software logging in persistent memory, when a log is shared by multiple threads. Even if a persistent memory system issues `clwb` instructions in each transaction, a context switch by the OS can occur before the `clwb` instruction executes. This context switch interrupts the control flow of transactions and diverts the program to other threads. This reintroduces the aforementioned issue of prematurely overwriting the records in a filled log. Implementing per-thread logs can mitigate this risk. However, doing so can introduce new persistent memory API and complicates recovery.

These inefficiencies expose the drawbacks of undo+redo logging in software and warrants a hardware solution.

3.2 Designing Undo+Redo Logging in Hardware

To address the challenges, this thesis proposes a hardware undo+redo logging design, consisting of Hardware Logging (HWL) and cache Force Write-Back (FWB) mechanisms. This section describes the design principles. Section 3.3 describes detailed implementation methods

and the required software support.

3.2.1 Assumptions and Architecture Overview

Figure 3.3(a) depicts an overview of the processor and memory architecture. The figure also shows the circular log structure in NVRAM. All processor components are completely volatile. This thesis uses write-back, write-allocate caches common to processors. This design supports hybrid DRAM+NVRAM for main memory, deployed on the processor-memory bus with separate memory controllers [135, 146]. However, this thesis focuses on persistent data updates to NVRAM.

Failure Model. Data in DRAM and caches, but not in NVRAM, are lost across system reboots. This design focuses on maintaining persistence of user-defined critical data stored in NVRAM. After failures, the system can recover this data by replaying the log in NVRAM. DRAM is used to store data without persistence [135, 146].

Persistent Memory Transactions. Like prior work in persistent memory [72, 135], this thesis uses persistent memory "transactions" as a software abstraction to indicate regions of memory that are persistent. Persistent memory writes require a persistence guarantee. Figure 3.2 illustrates a simple code example of a persistent memory transaction implemented with logging (Figure 3.2(a)), and with this design (Figure 3.2(b)). The transaction defines *object A* as critical data that needs persistence guarantee. Unlike most logging-based persistent memory transactions, transactions in this design eliminate explicit logging functions, cache forced write-back instructions, and memory barrier instructions. Section 3.3 describes the software interface design.

Uncacheable Logs in the NVRAM. For this design, this thesis uses a single-consumer, single-producer Lamport circular structure [75] for the log. The supported system software can allocate and truncate the log (Section 3.3). The hardware mechanisms append the log. A circular log structure is used because it allows simultaneous appends and truncates without locking [75,135]. Figure 3.3(a) shows that log records maintain undo and redo information of a single update (e.g., *store* A_1). In addition to the undo (A_1) and redo (A_1') values, log records also contain the following fields: a 16-bit transaction ID, an 8-bit thread ID, a 48-bit physical address of the data, and a *torn bit*. A *torn bit* per log entry is used to indicate the update is complete [135]. Torn bits have the same value for all entries in one pass over the log, but reverses when a log entry

is overwritten. Thus, completely-written log records all have the same torn bit value, while incomplete entries have mixed values [135]. The log must accommodate all write requests of undo+redo.

The log is typically used during system recovery, and rarely reused during application execution. Additionally, log updates must arrive in NVRAM in store-order. Therefore, this design makes the log *uncacheable*. This is in line with most prior works, in which log updates are written directly into a write-combine buffer (WCB) [135, 148] that coalesces multiple stores to the same cache line.

3.2.2 Hardware Logging (HWL)

The goal of the Hardware Logging (HWL) mechanism is to enable feasible undo+redo logging of persistent data in the microarchitecture. HWL also relaxes ordering constraints on caching in a manner that neither undo nor redo logging can. Furthermore, the HWL design leverages information naturally available in the cache hierarchy but not to the programmer or software. It does so without the performance overhead of unnecessary data movement or executing logging, cache force-write-back, or memory barrier instructions in pipeline.

Leveraging Existing Undo+Redo Information in Caches. Most processor caches use write-back, write-allocate caching policies [108]. On a write hit, a cache only updates the cache line in the hitting level with the new values. A dirty bit in the cache tag indicates cache values are modified but not committed to memory. On a write miss, the write-allocate (also called fetch-on-write) policy requires the cache to first load (i.e., allocate) the entire missing cache line before writing new values to it. HWL leverages the write-back, write-allocate caching policies to feasibly enable undo+redo logging in persistent memory. HWL automatically triggers a log update on a persistent write in hardware. HWL records both redo and undo information in the log entry in NVRAM (shown in Figure 3.2(b)). It gets the redo data from the currently in-flight write operation itself. It then gets the undo data from the write request's corresponding write-allocated cache line. If the write request hits in the L1 cache, HWL reads the old value before overwriting the cache line and use that for the undo log. If the write request misses in L1 cache, that cache line must first be allocated anyway, at which point HWL gets the undo data in a similar manner. The log entry, consisting of a transaction ID, thread, the address of the write, and undo and redo values, is written out to the circular log in NVRAM using the head and tail

pointers. These pointers are maintained in special registers described in Section 3.3.

Inherent Ordering Guarantee Between the Log and Data. This undo+redo hardware logging design does not require explicit memory barriers to enforce that undo log updates arrive at NVRAM before its corresponding working data. The ordering is naturally ensured by how HWL performs the undo logging and working data updates. This includes i) the uncached log updates and cached working data updates, and ii) store-granular undo logging. The working data writes must traverse the cache hierarchy, but the uncacheable undo log updates do not. Furthermore, HWL also provides an optional volatile log buffer in the processor, similar to the write-combining buffers in commodity processor design, that coalesces the log updates. The number of log buffer entries is configured based on cache access latency. Specifically, log updates certainly write out of the log buffer before a cached store writes out of the cache hierarchy. Section 3.3.3 and Section 3.5 further discuss and evaluate this log buffer.

3.2.3 Decoupling Cache FWBs and Transaction Execution

Writes are seemingly persistent once their logs are written to NVRAM. In fact, a transaction can commit once logging of that transaction is completed. However, this does not guarantee data persistence because of the circular structure of the log in NVRAM (Section 3.1.1). However, inserting cache write-back instructions (such as `clflush` and `clwb`) in software can impose substantial performance overhead (Section 3.1.1). This further complicates data persistence support in multithreading (Section 3.1.3).

This design eliminates the need for forced write-back instructions and guarantee persistence in multithreaded applications by designing a cache Force-Write-Back (FWB) mechanism in hardware. FWB is decoupled from the execution of each transaction. Hardware uses FWB to force certain cache blocks to write-back when necessary. FWB introduces a *force write-back* bit (*fwb*) alongside the tag and dirty bit of each cache line. A finite state machine in each cache block (Section 3.3.4) maintains the state using the *fwb* and *dirty* bits. Caches already maintain the *dirty* bit: a cache line update sets the bit and a cache eviction (write-back) resets it. A cache controller maintains the *fwb* bit by scanning cache lines periodically. On the first scan, it sets the *fwb* bit in dirty cache blocks if unset. On the second scan, it forces write-backs in all cache lines with $\{fwb, dirty\} = \{1, 1\}$. If the *dirty* bit ever gets reset for any reason, the *fwb* bit also resets and no forced write-back occurs.

The FWB design is also decoupled from software multithreading mechanisms. As such, this mechanism is impervious to software context switch interruptions. That is, when the OS requires the CPU to context switch, hardware waits until ongoing cache write-backs complete. The frequency of the forced write-backs can vary. However, forced write-backs must be faster than the rate at which log entries with uncommitted persistent updates are overwritten in the circular log. In fact, force write-back frequency (associated with the scanning frequency) is determined based on the log size and the NVRAM write bandwidth (discussed in Section 3.3.4). The evaluation shows the frequency determination (Section 3.5).

3.2.4 Instant Transaction Commits

Previous designs require software or hardware memory barriers (and/or cache force-write-backs) at transaction commits to enforce write ordering of log updates (or persistent data) into NVRAM across consecutive transactions [85, 146]. Instead, this design gives transaction commits a "free ride". That is, no explicit instructions are needed. The hardware logging mechanisms also naturally enforce the order of intra- and inter-transaction log updates: log updates issue in the order of writes to corresponding working data. The log updates write into NVRAM in the order they are issued (the log buffer is a FIFO). Therefore, log updates of subsequent transactions can only be written into NVRAM after current log updates are written and committed.

3.2.5 Putting It All Together

Figure 3.3(b) and (c) illustrate how the hardware logging works. Hardware treats all writes encompassed in persistent transactions (e.g., `write A` in the transaction delimited by `tx_begin` and `tx_commit` in Figure 3.2(b)) as persistent writes. Those writes invoke the HWL and FWB mechanisms. They work together as follows. Note that log updates go directly to the WCB or NVRAM if the system does not adopt the log buffer.

The processor sends writes of data object A (a variable or other data structure), consisting of new values of one or more cache lines $\{A'_1, A'_2, ...\}$, to the L1 cache. Upon updating an L1 cache line (e.g., from old value A_1 to a new value A'_1):

1. Write the new value (redo) into the cache line (❶).

(a) If the update is the first cache line update of data object A, the HWL mechanism (which has the transaction ID and the address of A from the CPU) writes a log record header into the log buffer.

(b) Otherwise, the HWL mechanism writes the new value (e.g., A_1') into the log buffer.

2. Obtain the undo data from the old value in the cache line (❷). This step runs parallel to Step-1.

   (a) If the cache line write request hits in L1 (Figure 3.3(b)), the L1 cache controller immediately extracts the old value (e.g., A_1) from the cache line before writing the new value. The cache controller reads the old value from the hitting line out of the cache read port and writes it into the log buffer in the Step-3. No additional read instruction is necessary.

   (b) If the write request misses in the L1 cache (Figure 3.3(c)), the cache hierarchy must write-allocate that cache block as is standard. The cache controller at a lower-level cache that owns that cache line extracts the old value (e.g., A_1). The cache controller sends the extracted old value to the log buffer in Step-3.

3. Update the undo information of the cache line: the cache controller writes the old value of the cache line (e.g., A_1) to the log buffer (❸).

4. The L1 cache controller updates the cache line in the L1 cache (❹). The cache line can be evicted via standard cache eviction policies without being subjected to data persistence constraints. Additionally, the log buffer is small enough to guarantee that log updates traverse through the log buffer faster than the cache line traverses the cache hierarchy (Section 3.3.4). Therefore, this step occurs without waiting for the corresponding log entries to arrive in NVRAM.

5. The memory controller evicts the log buffer entries to NVRAM in a FIFO manner (❹). This step is independent from other steps.

6. Repeat Step-1-(b) through 5 if the data object A consists of multiple cache line writes. The log buffer coalesces the log updates of any writes to the same cache line.

7. After log entries of all the writes in the transaction are issued, the transaction can commit (❺).

8. Persistent working data updates remain cached until they are written back to NVRAM by either normal eviction or cache FWB unit.

3.2.6 Discussion

Types of Logging. Systems with non-volatile memory can adopt centralized [63] or distributed (e.g., per-thread) logs [56, 140]. Distributed logs can be more scalable than centralized logs in large systems from software's perspective. Undo+redo hardware logging works with either type of logs. With centralized logging, each log record needs to maintain a thread ID, while distributed logs do not need to maintain this information in log records. With a centralized log, the hardware design effectively reduces the software overhead and can substantially improve system performance with real persistent memory workloads as show in the experiments. In addition, this design also allows systems to adopt alternative formats of distributed logs. For example, the physical address space can be partitioned into multiple regions and maintain a log per memory region.

NVRAM Capacity Utilization. Storing undo+redo log can consume more NVRAM space than either undo or redo alone. For its log, this design uses a fixed-size circular buffer rather than doubling any previous undo or redo log implementation. The log size trades off with the frequency of the cache FWB (Section 3.3). The software support discussed in Section 3.3.1 allow users to determine the size of the log. The FWB mechanism will adjust the frequency accordingly to ensure data persistence.

Lifetime of NVRAM Main Memory. The lifetime of the log region is not an issue. Suppose a log has 64K entries (4MB) and NVRAM (assuming phase-change memory) has a 200 ns write latency. Each entry will be overwritten once every 64K $\times$ 200 ns. If NVRAM endurance is 10^8 writes, a cell, even statically allocated to the log, will take 15 days to wear out, which is plenty of time for conventional NVRAM wear-leveling schemes to trigger [114, 115, 151]. In addition, this scheme has two impacts on overall NVRAM lifetime: logging normally leads to write amplification, but the design in this thesis improves NVRAM lifetime because the caches coalesce writes. The overall impact is likely slightly negative. However, wear-leveling will trigger before any damage occurs.

```
void persistent_update( int threadid )
{
  tx_begin( threadid );
  // Persistent data updates
  write A[threadid];
  tx_commit();
}
// ...
int main()
{
  // Executes one persistent
  // transaction per thread
  for ( int i = 0; i < nthreads; i++  )
    thread t( persistent_update, i );
}
```

Figure 3.4: Pseudocode example for `tx_begin` and `tx_commit`, where thread ID is transaction ID to perform one persistent transaction per thread.

3.3 Implementation Details

This section describes the implementation details of the hardware logging design and its overhead. Section 3.2.6 covered the impact of NVRAM space consumption, lifetime, and endurance.

3.3.1 Software Support

This design has software support for defining persistent memory transactions, allocating and truncating the circular log in NVRAM, and reserving a special character as the log header indicator.

Transaction Interface. This design uses a pair of transaction functions, `tx_begin( txid )` and `tx_commit()`, that define transactions which do persistent writes in the program. It can use `txid` to provide the transaction ID information used by the HWL mechanism. This ID is groups writes from the same transaction. This transaction interface has been used by numerous previous persistent memory designs [30, 146]. Figure 3.4 shows an example of multithreaded pseudocode with the transaction functions.

System Library Functions Maintain the Log. The HWL mechanism performs log updates, while the system software maintains the log structure. In particular, it uses system library functions, `log_create()` and `log_truncate()` (similar to functions used in prior work [135]), to allocate and truncate the log, respectively. The system software sets the log size. The memory controller obtains log maintenance information by reading special registers (Section 3.3.2), indicating the head and tail pointers of the log. Furthermore, a single transaction that exceeds the originally allocated log size can corrupt persistent data. There are two options to prevent overflows: 1) The `log_create()` function allocates a large-enough log by reading the maximum transaction size from the program interface (e.g., `#define MAX_TX_SIZE N`); 2) An additional library function `log_grow()` allocates additional log regions when the log is filled by an uncommitted transaction.

3.3.2 Special Registers

The `txid` argument from `tx_begin()` translates into an 8-bit unsigned integer (a *physical* transaction ID) stored in a special register in the processor. Because the transaction IDs group writes of the same transactions, the design can simply pick a not-in-use physical transaction ID to represent a newly received `txid`. An 8-bit length can accommodate 256 unique active persistent memory transactions at a time. A physical transaction ID can be reused after the transaction commits.

This design also uses two 64-bit special registers to store the head and tail pointers of the log. The system library initializes the pointer values when allocating the log using `log_create()`. During log updates, the memory controller and `log_truncate()` function update the pointers. If `log_grow()` is used, it employs additional registers to store the head and tail pointers of newly allocated log regions and an indicator of the active log region.

3.3.3 An Optional Volatile Log Buffer

To improve performance of log updates to NVRAM, this design provides an optional log buffer (a volatile FIFO, similar to WCB) in the memory controller to buffer and coalesce log updates. This log buffer is not required for ensuring data persistence, but only for performance optimization.

Data persistence requires that log records arrive at NVRAM before the corresponding

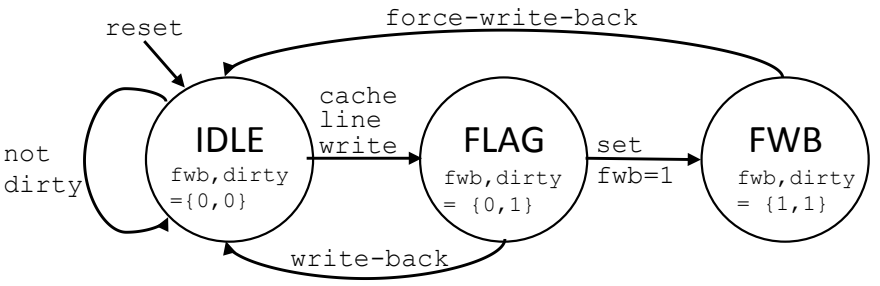


Figure 3.5: State machine in cache controller for FWB.

cache line with the working data. Without the log buffer, log updates are directly forced to the NVRAM bus without buffering in the processor. Choosing to adopt a log buffer with N entries means a log entry will take N cycles to reach the NVRAM bus. A data store sent to the L1 cache takes at least the latency (cycles) of all levels of cache access and memory controller queues before reaching the NVRAM bus. The the minimum value of this latency is known at design time. Therefore, it is guaranteed that log updates arrive at the NVRAM bus before the corresponding data stores by designing N to be smaller than the minimum number of cycles for a data store to traverse through the cache hierarchy. Section 3.5 evaluates the bound of N and system performance across various log buffer sizes based on the used system configurations.

3.3.4 Cache Modifications

Implementing the cache force write-back scheme requires adding one `fwb` bit to the tag of each cache line, alongside the *dirty* bit as in conventional cache implementations. FWB maintain three states (IDLE, FLAG, and FWB) for each cache block using these state bits.

Cache Block State Transition. Figure 3.5 shows the finite-state machine for FWB, implemented in the cache controller of each level. When an application begins executing, cache controllers initialize (reset) each cache line to the IDLE state by setting `fwb` bit to 0. Standard cache implementation also initializes *dirty* and *valid* bits to 0. During application execution, baseline cache controllers naturally set the *dirty* and *valid* bits to 1 whenever a cache line is written and reset the *dirty* bit back to 0 after the cache line is written back to a lower level (typically on eviction). To implement the state machine, the cache controllers periodically scan the valid, dirty, and `fwb` bits of each cache line and performs the following.

- A cache line with $\{fwb, dirty\} = \{0, 0\}$ is in IDLE state; the cache controller does nothing to those cache lines;
- A cache line with $\{fwb, dirty\} = \{0, 1\}$ is in the FLAG state; the cache controller sets the `fwb` bit to 1. This indicates that the cache line needs a write-back during the next scanning iteration if it is still in the cache.
- A cache line with $\{fwb, dirty\} = \{1, 1\}$ is in FWB state; the cache controller force writes-back this line. After the forced write-back, the cache controller changes the line back to IDLE state by resetting $\{fwb, dirty\} = \{0, 0\}$.
- If a cache line is evicted from the cache at any point, the cache controller resets its state to IDLE.

Determining the Cache FWB Frequency. The tag scanning frequency determines the frequency of the cache force write-back operations. The FWB must occur as frequently as to ensure that the working data is written back to NVRAM before its log records are overwritten by newer updates. As a result, the more frequent the write requests, the more frequent the log will be overwritten. The larger the log, the less frequent the log will be overwritten. Therefore, the scanning frequency is determined by the maximum log update frequency (bounded by NVRAM write bandwidth since applications cannot write to the NVRAM faster than its bandwidth) and log size (see the sensitivity study in Section 3.5). To accommodate large cache sizes with low scanning performance overhead, the size of the log can also grow to reduce the scanning frequency accordingly.

3.3.5 Summary of Hardware Overhead

Table 3.1 presents the hardware overhead of the implemented design in the processor. Note that these values may vary depending on the native processor and ISA. This implementation assumes a 64-bit machine, hence why the circular log head and tail pointers are 8 bytes. Only half of these bytes are required in a 32-bit machine. The size of the log buffer varies based on the size of the cache line. The size of the overhead needed for the `fwb` state varies on the total number of cache lines at all levels of cache. This is much lower than previous studies that track transaction information in cache tags [146]. The numbers in the table were computed based on the specifications of all the system caches described in Section 3.4.

Mechanism	Logic Type	Size
Transaction ID register	flip-flops	1 Byte
Log head pointer register	flip-flops	8 Bytes
Log tail pointer register	flip-flops	8 Bytes
Log buffer (optional)	SRAM	964 Bytes
Fwb tag bit	SRAM	768 Bytes

Table 3.1: Summary of major hardware overhead.

Note that these are major state logic components on-chip. The undo+redo logging design also also requires additional gates for logic operations. However, these gates are primarily small and medium-sized gates, on the same complexity level as a multiplexer or decoder.

3.3.6 Recovery

The steps of recovering the persistent data in systems that adopt this design are outlined as follows:

Step 1: Following a power failure, the first step is to obtain the head and tail pointers of the log in NVRAM. These pointers are part of the log structure. They allow systems to correctly order the log entries. Only one centralized circular log is used for all transactions for all threads.

Step 2: The system recovery handler fetches log entries from NVRAM and use the address, old value, and new value fields to generate writes to NVRAM to the addresses specified. The addresses are maintained via page table in NVRAM. Writes that did not commit are identified by tracing back from the tail pointer. Log entries with mismatched values in NVRAM are considered non-committed. The address stored with each entry corresponds to the address of the persistent data member. Aside from the head and tail pointers, a *torn bit* is used to correctly order these writes [135]. Log entries with the same `txid` and torn bit are complete.

Step 3: The generated writes bypass the caches and go directly to NVRAM. The caches are volatile, so their states are reset and all generated writes on recovery are persistent. Therefore, they can bypass the caches without issue.

Step 4: The head and tail pointers of the circular log get updated for each generated persistent write. After all updates from the log are redone (or undone), the head and tail pointers of the log point to entries to be invalidated.

Processor	Similar to Intel Core i7 / 22 nm
Cores	4 cores, 2.5GHz, 2 threads/core
IL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency,
DL1 Cache	32KB, 8-way set-associative, 64B cache lines, 1.6ns latency,
L2 Cache	8MB, 16-way set-associative, 64B cache lines, 4.4ns latency
Memory Controller	64-/64-entry read/write queues
NVRAM DIMM	8GB, 8 banks, 2KB row 36ns row-buffer hit, 100/300ns read/write row-buffer conflict [76].
Power and Energy	Processor: 149W (peak) NVRAM: row buffer read (write): 0.93 (1.02) pJ/bit, array read (write): 2.47 (16.82) pJ/bit [76]

Table 3.2: Processor and memory configurations.

3.4 Setup for Simulation

This thesis evaluates this design by implementing it in McSimA+ [12], a Pin-based [90] cycle-level multi-core simulator. The simulator is configured to model a multi-core out-of-order processor with NVRAM DIMM described in Table 3.2. This simulator also models additional memory traffic for logging and clwb instructions. The performance simulation results are fed into McPAT [82], a widely used architecture-level power and area modeling tool, to estimate processor dynamic energy consumption. The experiments modify the McPAT processor configuration to model the desired hardware modifications, including the components added to support HWL and FWB. The experiments also adopt phase-change memory parameters in the NVRAM DIMM [76]. Because all the performance numbers shown in Section 3.5 are relative (speedups), the same observations are valid for different NVRAM latency and access energy. This work focuses on improving persistent memory access so there is no DRAM access evaluation in these experiments.

The evaluation includes both microbenchmarks and real workloads in the experiments. The microbenchmarks repeatedly update persistent memory storing to different data structures including hash table, red-black tree, array, B+tree, and graph. These are data struc-

Name	Memory Footprint	Description
Hash [30]	256 MB	Searches for a value in an open-chain hash table. Insert if absent, remove if found.
RBTree [146]	256 MB	Searches for a value in a red-black tree. Insert if absent, remove if found
SPS [146]	1 GB	Random swaps between entries in a 1 GB vector of values.
BTree [22]	256 MB	Searches for a value in a B+ tree. Insert if absent, remove if found
SSCA2 [17]	16 MB	A transactional implementation of SSCA 2.2, performing several analyses of large, scale-free graph.

Table 3.3: A list of evaluated microbenchmarks.

tures widely used in storage systems [30]. Table 3.3 describes these benchmarks. These experiments use multiple versions of each benchmark and vary the data type between integers and strings within them. Data structures with integer elements pack less data (smaller than a cache line) per element, whereas those with strings require multiple cache lines per element. This allows exploring complex structures used in real-world applications. In these microbenchmarks, each transaction performs an insert, delete, or swap operation. The number of transactions is proportional to the data structure size, listed as "memory footprint" in Table 3.3. These benchmarks compile in native x86 and run on the McSimA+ simulator. Both singlethreaded and multithreaded versions of each benchmark are evaluated. In addition, this thesis evaluates the set of real workload benchmarks from the WHISPER persistent memory benchmark suite [102]. The benchmark suite incorporates various workloads, such as key-value stores, in-memory databases, and persistent data caching, which are likely to benefit from future persistent memory techniques.

3.5 Evaluating with Microbenchmarks and WHISPER

The design is evaluated in terms of transaction throughput, instruction per cycle (IPC), instruction count, NVRAM traffic, and dynamic energy consumption. The experiments compare among the following cases.

- **non-pers** – This uses NVRAM as a working memory without any data persistence or logging. This configuration yields an ideal yet unachievable performance for persistent memory systems [146].

- **unsafe-base** – This uses software logging without forced cache write-backs. As such, it does not guarantee data persistence (hence "unsafe"). Note that the dashed lines in the figures show the best case achieved between either redo or undo logging for that benchmark.

- **redo-clwb** and **undo-clwb** – Software redo and undo logging, respectively. These invoke the `clwb` instruction to force cache write-backs after persistent transactions.

- **hw-rlog** and **hw-ulog** – Hardware redo or undo logging with no persistence guarantee (like in unsafe-base). These show an extremely optimized performance of hardware undo or redo logging [146].

- **hwl** – This design includes undo+redo logging from the hardware logging (HWL) mechanism, but uses the `clwb` instruction to force cache write-backs.

- **fwb** – This is the full implementation of the hardware undo+redo logging design with both HWL and FWB.

3.5.1 Microbenchmark Results

This thesis makes the following major observations of the microbenchmark experiments and analyze the results. The benchmark evaluation is configured from single to eight threads. The prefixes of these results correspond to one (*-1t*), two (*-2t*), four (*-4t*), and eight (*-8t*) threads.

System Performance and Energy Consumption. Figure 3.6 and Figure 3.7 compare the transaction throughput and memory dynamic energy of each design. This thesis observes that processor dynamic energy is not significantly altered by different configurations. Therefore, only memory dynamic energy in the figure is shown. The figures illustrate that *hwl* alone improves system throughput and dynamic energy consumption, compared with software logging. Note that this design supports undo+redo logging, while the evaluated software logging mechanisms only support either undo or redo logging, not both. *Fwb* yields higher throughput and lower

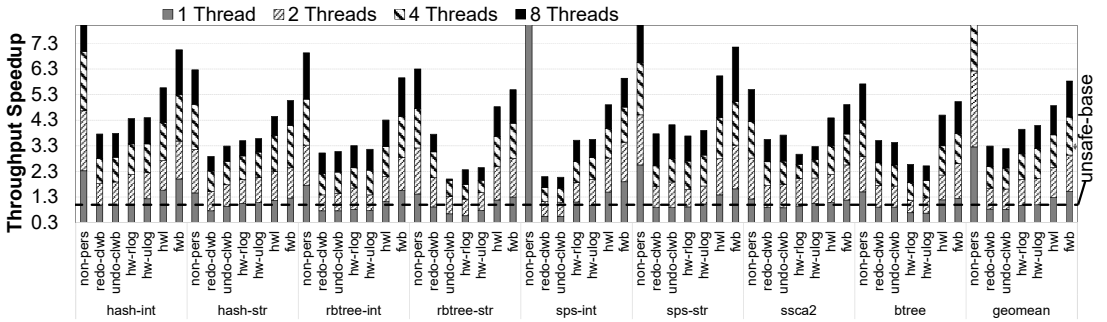


Figure 3.6: Transaction throughput speedup (higher is better), normalized to *unsafe-base*.

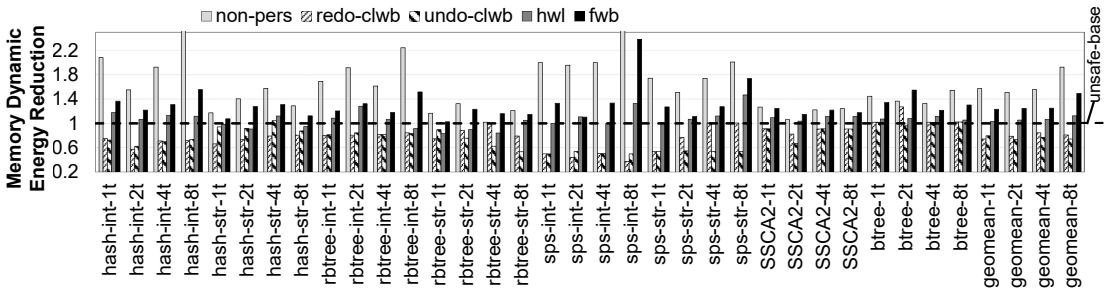


Figure 3.7: Dynamic energy reduction (higher is better), normalized to *unsafe-base* (dashed line).

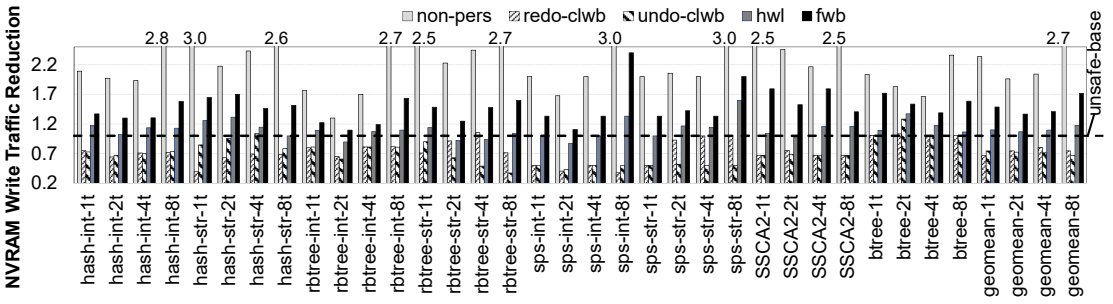


Figure 3.8: Memory write traffic reduction (higher is better), normalized to *unsafe-base* (dashed line).

energy consumption: overall, it improves throughput by 1.86× with one thread and 1.75× with eight threads, compared with the better of *redo-clwb* and *undo-clwb*. SSCA2 and BTree benchmarks generate less throughput and energy improvement over software logging. This is because SSCA2 and BTree use more complex data structures, where the overhead of manipulating the

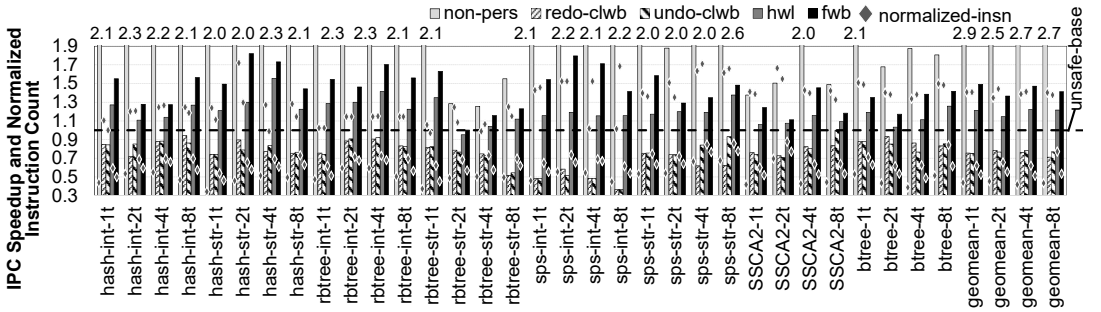


Figure 3.9: IPC speedup (higher is better) and instruction count (lower is better), normalized to *unsafe-base*.

data structures outweigh that of the log structures. Figure 3.8 shows that the undo+redo logging design substantially reduces NVRAM writes.

The figures also show that *unsafe-base*, *redo-clwb*, and *undo-clwb* significantly degrade throughput by up to 59% and impose up to 62% memory energy overhead compared with the ideal case *non-pers*. Undo+redo hardware logging brings system throughput back up. *Fwb* achieves 1.86× throughput, with only 6% processor-memory and 20% dynamic memory energy overhead, respectively. Furthermore, the design's performance and energy benefits over software logging remain as the number of threads increases.

IPC and Instruction Count. The evaluation also studies IPC number of executed instructions, shown in Figure 3.9. Overall, *hwl* and *fwb* significantly improve IPC over software logging. This appears promising because the figure shows the undo+redo hardware logging design executes much fewer instructions. Compared with *non-pers*, software logging imposes up to 2.5× the number of instructions executed. The *fwb* design only imposes a 30% instruction overhead.

Performance Sensitivity to Log Buffer Size. Section 3.3.3 discusses how the log buffer size is bounded by the data persistence requirement. The log updates must arrive at NVRAM before its corresponding working data updates. This bound is ≤15 entries based on the aforementioned processor configuration. Indeed, larger log buffers better improve throughput as studied using the *hash* benchmark (Figure 3.10(a)). An 8-entry log buffer improves system throughput by 10%; the implementation with a 15-entry log buffer improves throughput by 18%. Further increasing the log buffer size, which may no longer guarantee data persistence, additionally improves system throughput until reaching the NVRAM write bandwidth limitation (64 entries based on the NVRAM configuration). Note that the system throughput results with 128 and 256

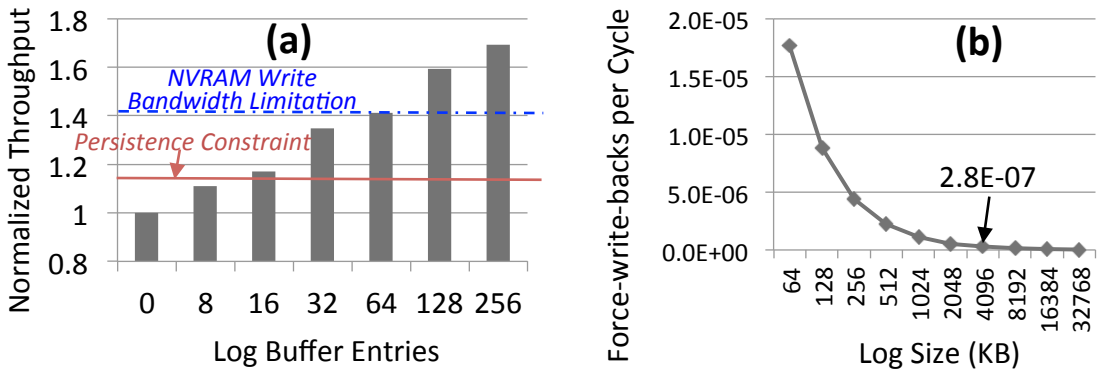


Figure 3.10: Sensitivity studies of (a) system throughput with varying log buffer sizes and (b) cache fwb frequency with various NVRAM log sizes.

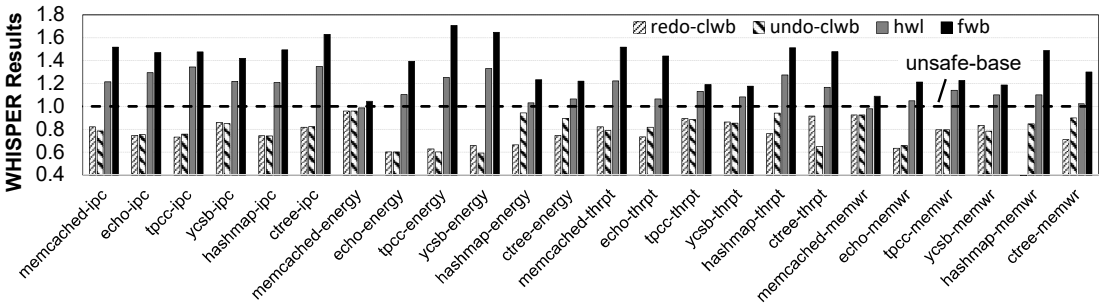


Figure 3.11: WHISPER benchmark results, including IPC, dynamic memory energy consumption, transaction throughput, and NVRAM write traffic, normalized to *unsafe-base* (the dashed line).

entries are generated assuming infinite NVRAM write bandwidth. Throughput also improves over baseline hardware logging *hw-rlog* and *hw-ulog*.

Relation Between FWB Frequency and Log Size. Section 3.3.4 discusses that the force write-back frequency is determined by the NVRAM write bandwidth and log size. With a given NVRAM write bandwidth, these experiments study the relation between the required FWB frequency and log size. Figure 3.10(b) shows that forced write-backs only need to occur every three million cycles with a 4MB log. As a result, the fwb tag scanning only introduces 3.6% performance overhead with a 8MB cache.

3.5.2 WHISPER Results

Compared with microbenchmarks, there are even more promising performance and energy improvements in real persistent memory workloads in the WHISPER benchmark suite with large data sets (Figure 3.11). Among the WHISPER benchmarks, ctree and hashmap benchmarks accurately correspond to and reflect the results achieved in the microbenchmarks due to their similarities. Although the magnitude of improvement vary, the undo+redo logging design leads to much higher performance, lower energy, and lower NVRAM traffic than the baselines. Compared with *redo-clwb* and *undo-clwb*, the undo+redo logging design significantly reduces the dynamic memory energy consumption of tpcc and ycsb due to the high write intensity in these workloads. Overall, undo+redo logging (*fwb*) achieves up to 2.7× the throughput of the best case in *redo-clwb* and *undo-clwb*. This is also within 73% of *non-pers* throughput of the same benchmarks. In addition, this design achieves up to a 2.43× reduction in dynamic memory over the baselines.

3.6 Discussion on RISC-V and RDMA Networking

This chapter has shown the high potential for hardware-based persistent memory design. Although promising, there is more that can be done to make hardware undo+redo logging fully robust. This section talks about two of those ways. First, implementing the design and expanding persistent memory support to RISC-V should enable many avenues of continued work. Second, supporting remote persistent requests via RDMA or memory networks.

3.6.1 RISC-V Support for Persistent Memory

Most persistent memory research is done using x86 due to extensive support in its instruction set [58]. Additionally, Intel has made a major push with development of its own non-volatile memory device: 3D Xpoint memory [59]. In particular, RISC-V [141] is widely used in academia but also more recently in industry-based research. Both academia and industry are investing in persistent memory and developing its technology, with examples shown earlier in this chapter. However, almost no support or work is being done for persistent memory using RISC-V. This is despite the fact that RISC-V is quickly growing in popularity and acceptance in both domains.

This thesis additionally proposes changes for RISC-V to support persistent memory. Current work is being done to fully integrate persistent memory and logging into a RISC-V system running on an FPGA as proof of concept. This implementation would enable identifying key challenges and optimizations for persistent memory not found on other ISAs. It also introduces new avenues of research into persistent memory using different architecture. Additionally, it would make RISC-V compatible with existing persistent memory work including benchmarks, file systems, and logging mechanisms.

In this chapter, as previously noted, the hardware undo+redo logging design is implemented in McSimA+ [12], a high-level C++ simulator and pin tool. Despite the hardware-intensive nature of the design, the limitations of this simulator and its ISA prevents a full study on the hardware complexity and challenges of hardware undo+redo logging. This would not be the case with RISC-V, whose current development includes chips and their RTL code for direct modification and design. In particular, this thesis does implement hardware undo+redo logging in the RTL design of the BOOM chip. This is accompanied by a design flow for including necessary changes to the system and to synthesize it on an FPGA for evaluation and experiments. This is a work in progress, but the ongoing work and planned future work in RISC-V are outlined in Section 6.2.

3.6.2 A Holistic Approach from Memory Bus to RDMA Network

The design in this chapter considered the entire system for its approach to implementing efficient hardware undo+redo logging. This design modified critical parts of existing microarchitecture and added new mechanisms to achieve its goal. However, this thesis implemented and studied hardware undo+redo logging in a single system. The assumption is that the persistent memory requests originated in the local system. However, this thesis has explored the possibility of persistent memory requests originating remotely. Although the fundamentals of hardware undo+redo logging could be expanded to account for this, this thesis does not explore that aspect in depth.

However, the work in this thesis has motivated contribution to a paper focusing on persistence parellelism optimization [55]. Specifically, the contribution is in regards to the observation that paper makes that the Remote Direct Memory Access (RDMA) network data path was being severely under-utilized in persistent memory. That paper also observes that

both industry and academia are considering and adopting remote NVM systems. Those types of system systems rely on memory persistence through an RDMA network, also known as network persistence.

Supporting persistent memory access through a network is becoming increasingly important. The software community already acknowledges the potential of remote persistent memory systems to provide fast write replication for better system availability in storage workloads and data center applications. In fact, remote persistent memory can replace SSDs for metadata storage in some high performance systems. The techniques developed in this thesis for hardware undo+redo logging inspired the proposed design in that paper and the design principles supplemented it.

Involvement in that paper underscored the significant role of networks and networking in the performance of memory systems. As state-of-the-art applications increase in memory demand and push the limits of memory models, it is clear that networks require attention. This motivates the focus and design of the next chapter in this thesis on memory networks. That design approach is not exclusive to persistent memory, rather it universally encompasses all types of memories. Nonetheless, persistent memory systems can reap the benefits of that design which furthers the goals of this thesis.

Chapter 4

String Figure: A Scalable and Elastic Memory Network Architecture

Demand for server memory capacity and performance is rapidly increasing due to expanding working set sizes of modern applications, such as big data analytics, in-memory computing, deep learning, and server virtualization. One promising techniques to tackle this requirements is memory networking, whereby a server memory system consists of multiple 3D die-stacked memory nodes interconnected by a high-speed network. However, current memory network designs face substantial scalability and flexibility challenges. This includes (1) maintaining high throughput and low latency in large-scale memory networks at low hardware cost, (2) efficiently interconnecting an arbitrary number of memory nodes, and (3) supporting flexible memory network scale expansion and reduction without major modification of the memory network design or physical implementation.

To address the challenges, this thesis proposes String Figure[1], a high-throughput, elastic, and scalable memory network architecture. String Figure consists of (1) an algorithm to generate random topologies that achieve high network throughput and near-optimal path lengths in large-scale memory networks, (2) a hybrid routing protocol that employs a mix of computation and look up tables to reduce the overhead of both in routing, (3) a set of network reconfiguration mechanisms that allow both static and dynamic network expansion and reduction. The experiments in this thesis using RTL simulation demonstrate that String Figure can inter-

[1] String Figure is a game formed by connecting strings between fingers of multiple people. This memory network topology appears like a string figure formed using memory nodes.

connect over one thousand memory nodes with a shortest path length within five hops across various traffic patterns and real workloads. This chapter details the design of String Figure and how it achieves these goals.

4.1 The Case for Memory Networks

This section briefly discuss the limitations of conventional DDRx-based memory systems, opportunities with memory networks, and the challenges of scalable memory network design.

4.1.1 Limitations of Commodity Server Memory Architectures

Traditional server memory systems face substantial challenges in scaling up memory capacity due to cost and performance issues. Increasing the number of CPU sockets enables adding more memory channels with more DIMMs. However, extra CPU sockets, which are added for memory capacity rather than compute requirement, can substantially increase hardware cost in an economically infeasible manner. Prior studies show that increasing memory capacity from 2TB to 4TB by doubling the number of CPU sockets can lead to over $3\times$ increase in server system cost [145]. Commodity clusters allow developing scale-out memory systems, which distribute the working set of applications across multiple server nodes [34]. However, the scale-out approach can only accommodate a limited subset of use cases with data-parallel, light communication algorithms. In remaining applications, the scale-out solution either requires programmers to rewrite their software, or system architects to adopt a low-latency interconnect fabric. This shifts the memory scaling burden to the software and communication infrastructures. However, recent studies that explored disaggregated memory system design [83, 91] require substantial changes in the virtual machine system software, such as a hypervisor. The String Figure design is in line with recent industry and academic approaches on implementing disaggregated memory pool with memory fabric, such as Gen-Z [1] and memory-centric system integration [67, 68, 144].

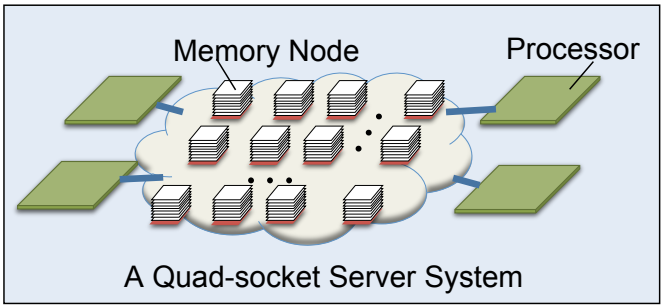


Figure 4.1: A disaggregated memory pool in a quad-socket server. Memory modules are interconnected by a memory network shared by four processors.

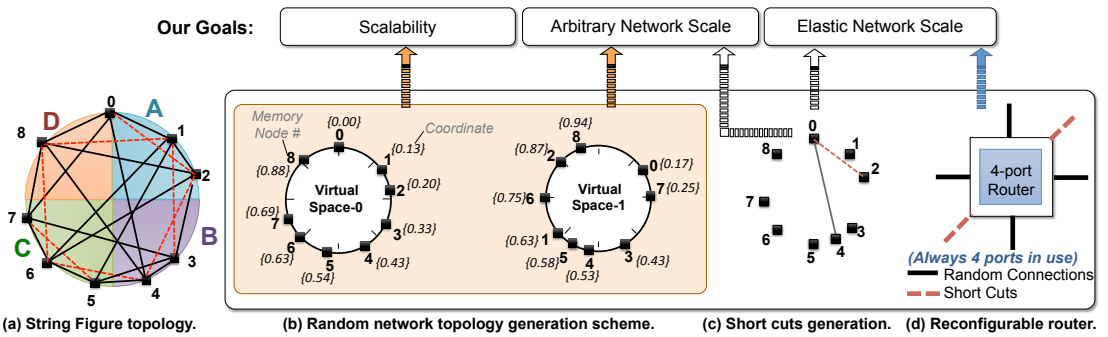


Figure 4.2: An example of String Figure topology design with nine memory nodes (stacks) and four-port routers. (a) String Figure topology. (b) Virtual space organization for random network generation. (c) Shortcuts generation for Node-0. (d) High-level design of a reconfigurable four-port router.

4.1.2 Memory Network

To address the memory capacity demand challenges, recent works introduce memory network design [67, 68]. A memory network consists of a set of 3D die-stacked memory nodes interconnected by a high-speed network. Figure 4.1 shows an example server system with four CPU sockets attached to a memory network. The processors can be connected to any edge memory nodes in the network. As 3D die-stacking technology is maturing, various 3D die-stacked memory designs are either already used in commodity systems or close to the market [2,5,15,94]. 3D die-stacked memory typically employs several DRAM dies on top of a logic die. For example, one type of die-stacked memory, Hybrid Memory Cube (HMC) [94, 101], offers 8GB capacity per memory stack with link speeds up to 30Gbps (versus 1.6Gbps supported by DDR4) and peak aggregate bandwidth of 320GB/s/link. Recent studies show that die-stacked

memory capacity is likely to scale to 16GB [112]. Memory network design has attracted both academic [66–68, 112, 144] and industry [1, 8, 130] efforts. Recent studies demonstrate that memory network can lead to more promising main memory expansion opportunities than traditional DIMM-based memory systems.

4.1.3 Challenges of Memory Network Design

Memory networks are constructed by interconnecting a large number of memory nodes. Therefore, performance and scalability of the interconnect network are essential to memory network performance and capacity scaling. Previous studies investigated how to best adopt traditional Network-on-Chip (NoC) topologies and protocols in memory network design, such as mesh, ring, crossbar, chain, skip list, butterfly, and tree [67, 68, 112, 144]. However, traditional memory network topologies can lead to substantial scalability challenges. In fact, most previous memory network designs can only interconnect tens of memory modes shared by all CPU sockets [67, 68, 112, 144]. To support terabytes of memory capacity, hundreds or over one thousand memory nodes need to interconnect with high network throughput and low access latency. As such, this thesis motivate this work with the following scalability and flexibility challenges that are not effectively addressed in prior works.

Network Scalability. As shown in recent studies, the shortest path lengths of various traditional memory network topologies can substantially increase with large-scale memory networks [112]. Instead, topologies used in data centers, such as Flattened Bufferfly [70], Dragonfly [69], Fat-Tree [14], and Jellyfish [122], can offer high bisection bandwidth and short routing path lengths in large-scale networks. However, these topologies are not directly applicable to memory network due to following reasons. First, data center networks adopt stand-alone network switches with rich link and storage resources. Second, most of these topologies require continuously increased router ports as the network scales up [14, 69, 70]. Finally, routing with most data center network topologies [122] requires large routing tables to store global routing information; the forwarding state cannot be aggregated. These issues hamper the memory networks from adopting data center network topologies in memory networks because of limited storage resources in on-chip routers and the high-bandwidth memory access requirement of in-memory applications. As a result, neither traditional memory networks nor data center network topologies can efficiently meet the scalability requirement of memory networks.

Arbitrary Network Scale. Many rigid network topologies require the number of routers and memory nodes to be specific numbers, such as a power of two. This reduces the flexibility of memory system scaling. Furthermore, these constraints on the network scale can increase the upgrade cost of memory systems and limit the potential of design reuse. For example, say there is the budget to purchase one more memory node to upgrade an existing memory network. It is difficult to upgrade because the rigid network topology only allows adding certain number of memory nodes (or none) to maintain the network scale as a power of two.

Elastic Network Size. Traditional memory systems allow users to reconfigure the memory capacity to a certain extent. For example, commodity computer systems typically reserve a certain number of memory slots for users to expand the memory capacity in the future. The same basic DIMM-based memory system designs in each DDRx generation are also shared across various design versions with different memory capacities. This flexibility allows future memory network designs to deliver cost-efficient memory system solutions. Furthermore, support for dynamic scaling up and down memory networks also enables efficient power management, by power gating off under-utilized links and idle memory nodes. Therefore, an elastic network size (static and dynamic network expansion and reduction) is a missing yet preferable feature for future memory networks.

4.2 String Figure Topology and Design

Overview. To achieve the design goals, this thesis proposes String Figure, a scalable, flexible, and high-performance memory network architecture. Figure 4.2 depicts an overview of these design goals and components – arrows in the figure map the proposed design components to the goals of this chapter. String Figure consists of three design principles. First, this thesis proposes an easy-to-implement random network topology construction algorithm that enables a) scalable memory network interconnecting large, arbitrary number of memory nodes with arbitrary number of router ports and b) support for elastic network scale. Second, this thesis proposes a compute+table hybrid routing scheme, which reduces the computation and storage overhead of routing large-scale networks by integrating a lightweight routing table with greediest computation-based routing mechanisms. Finally, this thesis proposes a network reconfiguration scheme, which enables elastic memory network scale. Beyond achieving the scalability goals, String Figure further enables memory access pattern aware performance optimization

and efficient memory network power management.

While String Figure is broadly applicable to a wide range of server memory systems, this thesis uses a working example throughout this chapter to make this proposal more concrete. This working example assumes a maximum 16TB memory system that consists of 1296 interconnected 3D die-stacked memory nodes shared by four CPU sockets (Figure 4.1). Each memory node has one router and is 8GB, with the same capacity and memory configuration parameters adopted in previous works [66, 112, 144]. Detailed baseline system configuration is described in Table 4.1.

4.2.1 Network Topology Construction Scheme

Prior studies in data center networks, such as Jellyfish [122], demonstrated that "sufficiently uniform random graphs" (i.e., graphs sampled sufficiently uniform-randomly from the space of all r-regular graphs) empirically have the scalability properties of random regular graphs [122] and achieve throughput within several percent to the upper bound on network throughput, at the scale of several thousand nodes. Such random topologies compare favorably against traditional fat-tree topologies, supporting a larger number of nodes at full throughput. However, most previous random topologies are developed for data center networks, which need to tolerate large forwarding state storage. Directly adopting these random topologies in memory networks, which is constrained by the storage capacity in routers and routing latency, can impose prohibitive scaling issues.

To address these challenges, this thesis proposes a novel network topology String Figure inspired by S2 [143] to enable scalable random topology in memory networks at low routing cost. The String Figure topology design also enables elastic memory network scale, i.e., flexible expansion and reduction of network scale. The topology consists of a basic balanced random topology and a set of shortcuts. The balanced random topology ensures scalability, interconnection of arbitrary number of memory nodes. The shortcuts provide extra links that maintain high network throughput, when the network scale is reconfigured (expansion or reduction) after being deployed (Section 4.2.3). Figure 4.2(a) illustrates an example topology interconnecting nine memory nodes, where each memory node has a four-port router. String Figure topology is generated offline before the memory network is deployed.

Balanced random topology generation algorithm. To simplify the topology construction pro-

(a) ALGORITHM 1. VIRTUAL SPACES CONSTRUCTION.

input: Number of memory nodes N
Number of ports per router p
output: {Coordinate x_i in Virtual Space$_j$ | i=0..N-1, j=0..L-1}

1 $L = \lfloor p/2 \rfloor$
2 **for** j = 0..L-1
3 **for** i = 0..N-1
4 x_{ij} = BalancedCoordinateGen(X_j)
5 X_j = sort(x_{0j}..x_{ij})
6 **return** {X_0..X_{L-1}}

(b) ALGORITHM 2. BALANCED COORDINATE GENERATION.

input: Coordinates of k memory nodes x_0..x_{k-1}
output: Coordinate of a new memory node x_n

1 **if** k = 0 **then return** RandomNumber(0, 1)
2 **if** k = 1
3 **then** $a \leftarrow x_0$, $b \leftarrow x_0 + 1$
4 **else**
5 $D(x_{r1}, x_{r2}) = min\{|x_{r1} - x_{r2}|, 1 - |x_{r1} - x_{r2}|\}$
6 find x_{r1}, x_{r2} among x_0..x_{k-1} such that
7 $x_{r1} < x_{r2}$ and $D(x_{r1}, x_{r2})$ is the smallest
8 **if** $x_{r2} - x_{r1} < 1/2$
9 **then** $a \leftarrow x_{r1}$, $b \leftarrow x_{r2}$
10 **else** $a \leftarrow x_{r2}$, $b \leftarrow x_{r1} + 1$
11 $x_n \leftarrow$ RandomNumber($a + 1/(3(k-1))$, $b - 1/(3(k-1))$)
12 **if** $x_n > 1$ **then** $x_n \leftarrow x_n - 1$
13 **return** x_n

Figure 4.3: Algorithms for generating (a) balanced random topologies and (b) the used balanced coordinate generation function BalancedCoordinateGen(), where D is circular distance defined in in the routing protocol.

cess for system developers, this thesis designs a topology generation algorithm, which answers two critical questions: (i) Randomness – how do we ensure that the generated networks are uniformly-random? (ii) Balance – how do we ensure balanced connections? Imbalanced connections are likely to increase congestion. Figure 4.3 illustrates the random topology generation algorithm used in String Figure. The example in Figure 4.2(b) explains this design. Inputs of the algorithm include the number of memory nodes N and the number of router ports p. This approach consists of four steps:

- Constructing L virtual spaces, where the number of virtual space $L = \lfloor \frac{p}{2} \rfloor$. For example, a memory network with four-port routers (not including the terminal port) will lead to maximum two virtual spaces: Space-0 and Space-1.

- Virtually (i.e., logically) distributing all the memory nodes in each virtual space with a random order. Random orders are generated by assigning random coordinates to memory nodes. For example, the coordinates of Node-2 is 0.20 and 0.87 in Space-0 and Space-1,

respectively.

- Interconnecting the neighboring memory nodes in each virtual space. For instance, Node-2 is connected with Node-1 and Node-3 in Space-0; it is also connected with Node-6 and Node-8.

- Interconnecting pairs of memory nodes with free ports remaining. For example, because Node-5 and Node-4 are connected in both spaces, Node-5 will have a free port left. Therefore, String Figure can additionally connect Node-5 with Node-3, which also has a free port. When multiple choices exist, String Figure selects the pairs of memory nodes with the longest distance.

The solid lines in Figure 4.2(a) illustrate an example of generated basic random topology. The String Figure topology generation algorithm only determines which nodes are interconnected. Both uni-directional and bi-directional connections are allowed (discussed in Section 4.3). The four router ports in this example are all used to connect to other memory nodes in the network. To connect to processors, each router has an additional port, i.e., each router in this example would have five ports in total. Processors can be attached to any subset memory nodes, or all of them (evaluated in Section 4.6).

Shortcuts generation. The goal of adding extra connections is to maintain high network throughput, when the memory network needs to scale down after it is deployed, e.g., by shutting down (power-gating [144]) routers and corresponding links (more details in Section 4.2.3). To achieve this goal, String Figure generates shortcuts for each memory node to its two and four hop neighbors – within short circular distance – distributed in Virtual Space-0 in a clockwise manner Figure 4.2(c) shows shortcuts generated for Node-0. Nodes only connect to a node with node id larger than itself. For example, String Figure does not connect Node-5 to Node-0, although Node-0 is Node-5's four-hop neighbor. As such, it limits the link and router hardware overhead by adding maximum two shortcut connections for each node. The connections not existing in the basic balanced random topology are added into the network (e.g., the red dash line between Node-0 and Node-2). Figure 4.2(a) depicts the final topology combining the basic random topology (black solid lines) and shortcuts (red dash lines). The rationale behind is two-fold. First, it demonstrate that one-, two-, and four-hop connections efficiently accommodate data access of big-data workloads, such as query in a distributed hash table [127]. Second, if

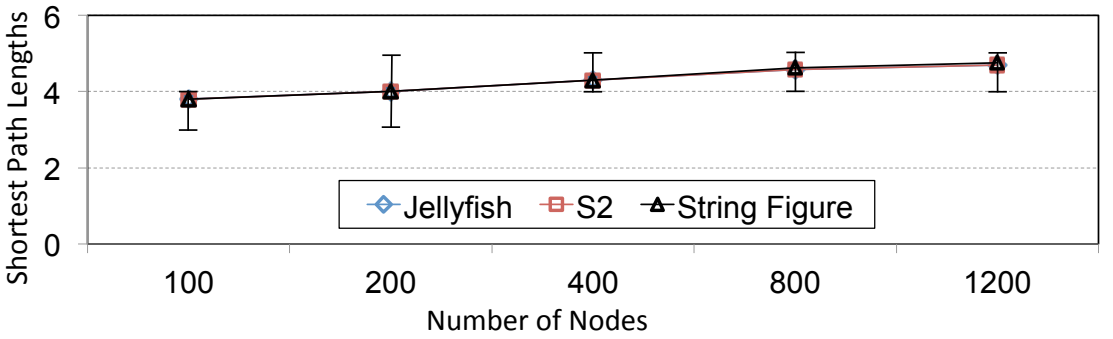


Figure 4.4: Comparison of shortest path lengths.

Node #	D in Space-0	D in Space-1	MD
7	0.49	0.62	0.49
0	0.20	0.70	0.20
3	0.13	0.44	0.13
6	0.43	0.12	0.12
8	0.68	0.07	0.07

(a)

Node#	Block.	Valid	Hop	Space#	Coordi.
0	1	1	0	0	0.00
2	1	1	0	0	0.20
...					
5	1	1	0	1	0.58
6	1	1	0	1	0.75
...					
3	1	1	1	0	0.33
8	1	1	1	0	0.88
...					

(b)

Figure 4.5: Greedy routing protocol. (a) Circular distances (D) and minimum circular distances to Node-2 (MD) from Node-7 and Node-7's neighbors. (b) Routing table entries (16 entries in total).

String Figure divides Virtual Space-0 into four sectors (A, B, C, and D in Figure 4.2(a)), the combination of the random topology and the shortcuts ensures that the memory network has direct connections between both short- and long-circular-distance nodes in every two sectors; when the network is scaled down, the shortcuts can maintain high throughput by fully utilize router ports.

Sufficiently uniform randomness of the topology. To show that String Figure provides sufficiently uniform random graphs (SURGs), this thesis compares the average shortest path length of the String Figure topology with Jellyfish [122] and S2, which are proved to offer SURG. Empirical results String Figure topology leads to similar average shortest path lengths with the same path length bounds across various network scales.

4.2.2 Routing Protocol

A desired routing protocol in memory network needs to be scalable, deadlock free, while fully utilizing the bandwidth of the network topology. Conventional routing schemes on random topologies employ k-shortest path routing with global information stored in the routing table (a look-up table) of each router [122]. Given a memory network with N memory nodes, the routing table size and routing algorithm complexity can be up to $O(NlogN)$ and $O(N^2logN)$. In order to maintain sub-linear increase of routing overhead – consisting of look-up table size and routing algorithm complexity – String Figure adopts a hybrid compute+table routing scheme. Section 4.3 discusses the deadlock avoidance mechanism used.

Greediest routing. To achieve high-performance routing with small routing table storage, String Figure employs a scalable greedy routing protocol, namely greediest routing [143]. Most previous greedy routing protocols only minimize distance to the destination in a single space. The basic random topology used in String Figure consists of multiple virtual spaces. Therefore, it needs a routing protocol that can identify the shortest distance to the destination among all neighbors in all virtual spaces. To this end, String Figure makes forwarding decisions by a fixed, small number of numerical distance computation and comparisons. This thesis defines circular distance (D) as the distance between two coordinates u and v in each virtual space:

$$D(x, y) = min\{|u - v|, 1 - |u - v|\}$$

The routing algorithm then calculates the minimum circular distance between two nodes as the following, given the two nodes with the set of coordinates in L virtual spaces $\overrightarrow{\mathrm{U}} =< u_1, u_2, ..., u_L >$ and $\overrightarrow{\mathrm{V}} =< v_1, v_2, ..., v_L >$.

$$MD(\overrightarrow{\mathrm{U}}, \overrightarrow{\mathrm{V}}) = min\{D(u_i, v_i)\}$$

Forwarding decision making: To forward a packet from Node-s to destination Node-t, the router in Node-s first selects a neighbor Node-w such that w minimizes $MD(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}})$ to the destination. The packet is then forwarded to Node-w. This process will continue until Node-w is the destination. For instance, Node-7 needs to send a packet to Node-2. Figure 4.5(a) shows the minimum circular distances to Node-2 from Node-7 and Node-7's one-hop neighbors in the example (Figure 4.2). Node-7 has four neighbors Node-0, 3, 6, and 8. Based on the

computation, Node-8 has the minimum MD from Node-7. The packet is then forwarded to Node-8. Node-8 will then determine the next stop for the packet by computing a new set of MDs based on its local routing table. The router in each memory node only maintains a small routing table that stores coordinates of its one- and two-hop neighbors in all virtual spaces (Figure 4.5(b)). To further reduce the routing path lengths, String Figure computes MD with both one- and two-hop neighbor information (based on the sensitivity studies in the results section) stored in the routing table in each router. Routing table implementation is discussed in Section 4.3. The forwarding decision can be made by a fixed, small number of numerical distance computation and comparisons; the decisions are made locally without link-state broadcast in the network wide.

Adaptive routing. By only storing two-hop neighbors in routing tables, the greediest routing algorithm does not guarantee shortest routing path. As such, String Figure offers path diversity across different virtual spaces, i.e., can have multiple paths satisfying the MD requirement. By leveraging path diversity, String Figure employs adaptive routing to reduce network congestion. This only divert the routing paths by tuning the first hop forwarding decision. With a packet to be sent from node s to destination t, Node-s can determine a set W of neighbors, such that for any $w \in W$, $MD(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}}) < MD(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$, where t is the destination. Node-s then selects one neighbor in W based on traffic load across s's router ports. A counter is used to track the number of packets waiting at each port to estimate the network traffic load on each outgoing link. At the first hop, the source router can select the links with lightly loaded port satisfying the greediest routing requirement, rather than a heavily loaded port with the queue filled by a user-defined threshold (e.g., 50%). This enforces that String Figure routing reduces the MD to the destination at every hop, eventually finding a loop-free path to the destination (Section 4.3).

4.2.3 Reconfigurable Memory Network

To achieve the goal of an elastic network scale, this thesis proposes a set of reconfigurable memory network mechanisms. String Figure naturally supports memory network scaling up and down at low performance and implementation cost, because it (i) allows arbitrary number of memory nodes, (ii) always fully utilizes the ports in each router, and (iii) only requires local routing table information to make routing decisions. This thesis also designs a topology switch (Figure 4.6) in the routers to reconfigure the links (details described in Section 4.3).

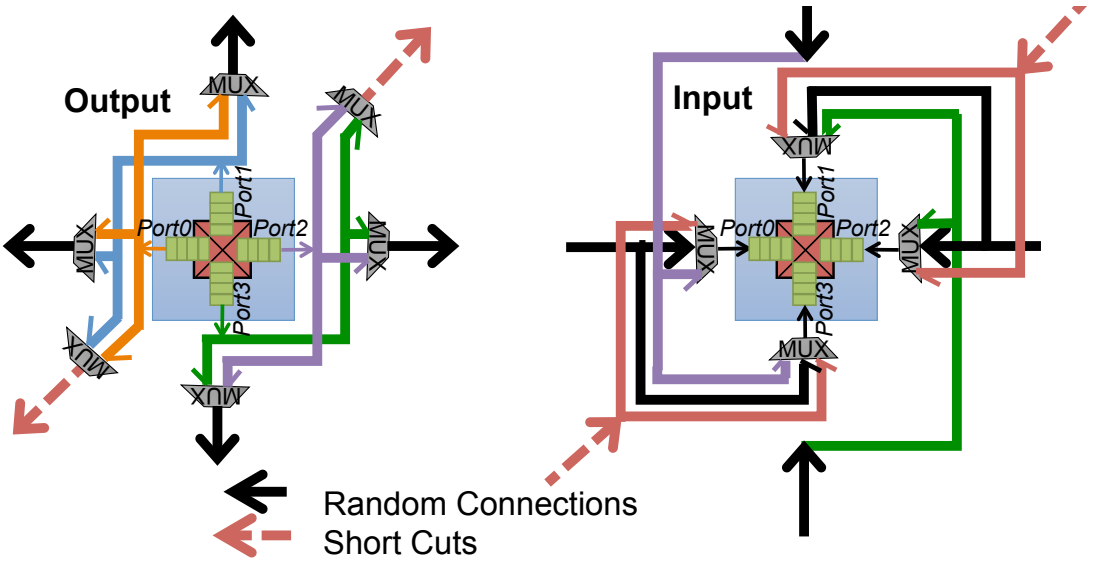


Figure 4.6: An example topology switch design.

Dynamic reconfiguration for power management. This design allows the memory network to dynamically scale up and down (e.g., by power gating routers and links) to enable efficient power management. Memory network power management needs to be carefully designed with most traditional topologies, such as meshes and trees. Otherwise, some nodes can be isolated and require a long wake-up latency on the order of several microseconds [144]. To address this issue, this thesis designs a dynamic reconfiguration mechanism that maintains high network throughput after turning off routers. This dynamic network scale reconfiguration involves four steps. First, it blocks corresponding routing table entries (Figure 4.5(b)) in associated routers. Second, it enables and disables certain connections between memory nodes. The shortcuts are used to maintain high network throughput after disabling certain links. Third, it validates and invalidates corresponding routing table entries in associated routers. Finally, it unblocks the corresponding routing table entries. The first and last steps ensure atomic network reconfiguration. For example, to turn off Node-1 in Figure 4.2(a), String Figure will disconnect Node-1 from its one-hop neighbors Node-0, Node-2, Node-5, and Node-6. In each of these nodes, it invalidates the routing table entries that store Node-1's coordinates. The shortcuts between Node-0 and Node-2 are enabled because both nodes now have one free port. The corresponding routing table entries are then modified for these two nodes, indicating that the original two-hop neighbors are now one-hop neighbors. Bringing Node-1 back into the network uses the same

steps but in reverse. Because the String Figure routing protocol maintains two-hop neighbors in each router, each update of routing table entries is simply flipping the blocking, valid, and hop# bits without needing to add or delete entries. Updates in different routers can be performed in parallel.

Static network expansion and reduction for design reuse. Design reuse can reduce the cost and effort of redesigning, re-fabricating, and upgrading systems. Specifically, design reuse allows system developers to reuse a a memory network design or fabrication across server memory systems with different capacity requirement. The previously outlined steps of the dynamic network reconfiguration are used while offline to enable network expansion and reduction. To support network expansion, system developers can implement a larger network size than currently needed and deploy the memory network with only a subset of memory nodes mounted. The excess nodes are "reserved" for future use. String Figure enables and validates corresponding links and routing table entries based on the mounted memory nodes. As such, network expansion does not require redesign or re-fabrication of the entire memory network. If memory nodes are interfaced through PCBs (e.g., HMC-style), the memory network can be expanded by mounting additional memory nodes on the PCB, followed by a link and routing table reconfiguration in the same way as dynamic reconfiguration. As a result, String Figure can reduce the cost and effort of re-fabricating PCBs. Network scale reduction is performed in reverse, by unmounting memory nodes. If the memory nodes are mounted on a silicon interposer (e.g., HBM-styled), chips need to be fabricated with added memory nodes by reusing the original memory network design. However, the design stays the same, substantially reducing non-recurring engineering (NRE) cost.

4.3 Router Design and Deadlock Avoidance

This section describes implementation details of String Figure, including deadlock avoidance, reconfigurable router design, and physical implementation.

4.3.1 Deadlock Avoidance

Two conditions must be met to avoid deadlocks in the String Figure topology. First, route paths must be loop-free from source to destination. Second, the network cannot have

cyclical resource dependencies whereby routers wait on each other to free up resources (like buffers). String Figure's greedy routing naturally ensures that route paths are loop-free. String Figure uses virtual channels to avoid deadlocks from resource dependencies.

Loop-free routing paths. String Figure ensures the routes between any source-destination pair are loop-free because it always routes greedily within our network topology. This is guaranteed by the *progressive and distance-reducing* property (Section 4.4) of our greedy routing protocol. Section 4.4 formally proves that packet routes are loop-free.

Avoiding deadlocks with virtual channels. String Figure adopts two virtual channels [21, 78, 92] to avoid deadlocks. Packets use one virtual channel when routing from a source of a lower space coordinate to a destination of a higher space coordinate; packets use the other virtual channel when routing from a source of higher space coordinate to a destination of a lower space coordinate. This avoids deadlocks because in the String Figure topology (which is not truly random), packets are only routed through to networks with a strictly increasing coordinate or a strictly decreasing coordinate; the only dependency is between the virtual channels in the router, which is insufficient to form cycles [21]. Whereas virtual channels can increase the required buffering, String Figure allows the number of router ports to remain constant as the network scales up. Therefore, the buffer size overhead is less of an issue compared with prior works [21, 53] (evaluated in Section 4.6).

4.3.2 Router Implementation and Reconfiguration

This thesis designs the router on each memory node to facilitate the routing table design, reconfigurable links, and counters for adaptive routing.

Routing table implementation. Figure 4.5(b) illustrates the routing table implementation. Each routing table stores information of its one- and two-hop neighbors, including $log_2 N$ memory node number, 1-bit blocking bit, 1-bit valid bit, 1-bit hop number ('0' for one-hop and '1' for two-hop), $\lceil log_2 \frac{p}{2} \rceil$ virtual space number, and 7-bit virtual coordinate. The routing table entries initialize accordingly, while the network topology generates. Once the network topology is generated, only the blocking, valid, and hop bits update during network reconfiguration. A memory node has maximum two one-hop neighbors in each virtual space; each of the one-hop neighbors has two one-hop neighbors of their own in each virtual space. As the maximum number of virtual spaces is half of the number of ports (p), each routing table has a maximum of

$p(p+1)$ entries.

Enabling link and topology reconfiguration with switches. The String Figure memory network reconfiguration requires connecting and disconnecting certain links between neighbor memory nodes. Its basic balanced random network topology already fully utilizes all router ports. However, each node also has at most two shortcut connections (Section 4.2.1). To accommodate the shortcuts, String Figure uses a switch to attach the two shortcut connections to two of the router ports at each node. Figure 4.6 shows the topology switch design. It is comprised of a set of multiplexers similar to prior reconfigurable NoC designs [61,128]. As a result, the topology switches allow selecting p (the number of router ports) connections out of all the random connections and shortcuts provided by the String Figure topology.

Tracking port utilization with packet counters. With adaptive routing, String Figure uses counters at each port to track the queue length at the port. The number of counter bits is $log_2 q$, where q is the number of queue entries of the port. The counter provides an additional variable for determining routing paths. It specifically tracks the congestion of each path by counting how often packets route to specific outputs ports. The routing algorithm uses these counter values to make smarter decisions. If an output port has too many packets routed to it, the algorithm detects this through counters and chooses alternate, yet still greedy, output ports to send the packet. This helps avoid congestion in the network and still achieve low latency, high throughput performance overall. These counters are reset after the network is reconfigured.

4.3.3 Physical Implementation

The goal of the physical implementation is to reduce both the area overhead and long wires in String Figure.

Bounded number of connections in the network. With the String Figure network topology, the number of connections coming out of each node is bounded by the number of router ports (p) and remains *constant*, independent of network scale (N). Each node has $\frac{p}{2}$ one-hop neighbors in the basic random topology and a maximum of two shortcuts (some generated shortcuts will overlap with connections in the basic random topology). Therefore, the total number of connections coming out of each node $C_{node} \leq \frac{p}{2} + 2$. For example, a memory node with an 8-port router only requires six connections per node. Given N memory nodes in total, the total number of required connections in the network $C_{network} \leq N \times (\frac{p}{2} + 2)$, which grows linearly

with the number of memory nodes.

Uni-directional versus bi-directional connections. Bi-directional connections allow packets to traverse the network both forward and backwards. Uni-directional connections typically have worse packet latency than their bi-directional counterparts, due to reduced path diversity. However, uni-directional networks have lower hardware and energy cost than bi-directional connections. The sensitivity studies in Section 4.6 demonstrate that uni-directional networks perform almost the same as bi-directional networks; their discrepancy diminishes with increasing number of nodes in the network. Therefore, String Figure uses uni-directional connections.

Memory node placement and wire routing. When building String Figure, memory nodes are placed in the memory network (on PCB or silicon interposer) as a 2D grid. The goal of memory node placement is to reduce long wires. Memory network implementations are constrained by wire lengths [3, 94]. For example, HBMs [3] (with a 7 mm dimension in HBM1 and 12 mm in HBM2) are implemented with interposers to support large-scale memory networks; previous works demonstrate that memory nodes can be clustered with MetaCubes [112] (i.e., clustered memory nodes integrated with an interposer), which is further interconnected with other interposer-integrated clusters. To achieve the goal of this memory network design, String Figure set two priority levels that prioritize the clustering of one-hop and two-hop neighbors. For example, it ensures that all one-hop neighbors are placed within ten grid distance with place and routing. The String Figure network topology also naturally supports MetaCube [112] architecture. It provides connections with various circular distances. As such, memory nodes are placed with short circular distances in the same MetaCubes. Inter-MetaCube links are implemented by connections with long circular distances.

Processor placement. The flexibility of String Figure topology and routing protocol allows attaching a processor to any one or multiple memory nodes. The router at each memory node has a local port connecting to the processor. As such, attaching a processor to multiple memory nodes can increase processor-memory bandwidth. By tuning traffic patterns of the synthetic workloads, the evaluation examines ways of injecting memory traffic from various locations, such as corner memory nodes, subset of memory nodes, random memory nodes, and all memory nodes.

4.4 Proofs and Lemmas Used For String Figure

This section formally presets the lemmas used in designing String Figure.

The following lemma formally proves the proposition on loop freedom by two lemmas similar to those derived in data center networks [143].

Lemma 1. *In a virtual space with a given a coordinate x, if a memory node s is not the node that has the shortest circular distance to x in this space, then s must have an adjacent router s', such that $D(x, x_{s'}) < D(x, x_s)$.*

Proof. 1. Let w be the node closest to x among all memory nodes in the virtual space.

2. The ring of this space is divided by s and x into two arcs. At least one of the arcs has a length no greater than $\frac{1}{2}$. Suppose that $\widehat{x_s, x}$ with length $l(\widehat{x_s, x})$. This means $D(x_s, x) = l(\widehat{x_s, x}) \leq \frac{1}{2}$.

3. If w is on $\widehat{x_s, x}$, let the arc between s and w be $\widehat{x_s, x_w}$.

   (a) If s has an adjacent node q with coordinate on $\widehat{x_s, x_w}$, then $l(\widehat{x_q, x}) < l(\widehat{x_s, x})$. Hence, $D(q, x) = l(\widehat{x_q, x}) < l(\widehat{x_s, x}) < D(x_s, x)$.

   (b) If s has no adjacent node on $\widehat{x_s, x_w}$, w is x's adjacent node. Hence, s has an adjacent node w, such that $D(x, x_w) < D(x, x_s)$.

4. If w is not on $\widehat{x_s, x}$, there is an arc $\widehat{x_s, x, x_w}$. For the arc $\widehat{x, x_w}$ on $\widehat{x_s, x, x_w}$, then $l(\widehat{x, x_w}) < l(\widehat{x_s, x})$. (Assuming to the contrary, if $l(\widehat{x, x_w}) \geq l(\widehat{x_s, x})$, then there cannot be $D(x, x_w) < D(x, x_s)$. There is contradiction.)

   (a) If s has an adjacent memory node q with coordinate on $\widehat{x_s, x, x_w}$, then $l(\widehat{x_q, x}) < l(\widehat{x_s, x}) \leq \frac{1}{2}$. Hence, $D(q, x) = l(\widehat{x_q, x}) < l(\widehat{x_s, x} = D(x_s, x)$.

   (b) If s has no adjacent memory node on $\widehat{x_s, x, x_w}$, w is x's adjacent node. Hence, s has an adjacent node w, such that $D(x, x_w) < D(x, x_s)$.

5. Combining (3) and (4), s always has an adjacent node s', such that $D(x, x_{s'}) < D(x, x_s)$.

■

Lemma 2. *Suppose the source and destination of a packet are routers s and t, respectively. Coordinates of the destination router in all virtual spaces are $\overrightarrow{\mathrm{X_t}}$. Let w be the router that has the minimum MD to t among all neighbors of s, then $MD(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}}) < MD(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.*

Proof.

1. Suppose the minimum circular distance between s and t is defined by their circular distance in the jth space, i.e., $D(x_{tj}, x_{sj}) = MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.
2. In the jth space, t is the memory node with the shortest circular distance to x_{tj}, which is $D(x_{tj}, x_{tj}) = 0$. Because $s \neq t$, s is not the node with the shortest circular distance to x_{tj}.
3. Based on Lemma 1, s has an adjacent memory node s', such that $D(x_{tj}, x_{s'j}) < D(x_{tj}, x_{sj})$.
4. Then, $MD_L(\overrightarrow{\mathrm{X_{s'}}}, \overrightarrow{\mathrm{X_t}}) \leq D(x_{tj}, x_{x'j}) < D(x_{tj}, x_{sj}) = MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.
5. Because w is the node that has the shortest MD to $\overrightarrow{\mathrm{X_t}}$ among all neighbors of s, then $MD_L(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}}) \leq MD_L(\overrightarrow{\mathrm{X_{s'}}}, \overrightarrow{\mathrm{X_t}} \leq MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$.

■

Lemma 2 states on a packet's route, if a router s is not the destination, it must find a neighbor whose MD is smaller than s's MD to the destination.

Proposition 3. *Greediest routing finds a loop-free path of a finite number of hops to a given destination in the String Figure topology.*

Proof.

1. Suppose memory node s receives a packet with destination node t. If $s = t$, then s is the destination. The packet arrives at the destination.
2. If $s \neq t$, according to Lemma 2, s will find a neighbor w, such that $MD_L(\overrightarrow{\mathrm{X_w}}, \overrightarrow{\mathrm{X_t}}) < MD_L(\overrightarrow{\mathrm{X_s}}, \overrightarrow{\mathrm{X_t}})$, and forward the packet to w.
3. The MD from the current memory node to the destination coordinates strictly reduces at each hop. Routing keeps making progress. Therefore, there is no routing loop.

■

Table 4.1: System configuration.

CPU	4 sockets; 2GHz; 64B cache-line size
Memory	up to 1296 memory nodes; 8GB per memory node (stack)
DRAM timing	tRCD=12ns, tCL=6ns, tRP=14ns, tRAS=33ns
CPU-memory channel	256 lanes in total (128 input lanes and 128 output lanes); 30Gbps per lane
SerDes delay	3.2ns SerDes latency (1.6ns each) per hop
Energy	Network: 5pJ/bit/hop; DRAM read/write: 12pJ/bit

Topology	**Number of Nodes (N), Number of Ports per Router (p)**											**Routing Scheme**
	N	**16**	**17**	**32**	**61**	**64**	**113**	**128**	**256**	**512**	**1024**	**1296**	
Distributed-Mesh (DM)/ Optimized DM (ODM)	p	4	N	4	N	4	N	4	4	4	4	4	Greedy + adaptive
Flattened Butterfly (FB)	p								20	24	31	33	Minimal + adaptive
Adapted FB (AFB)	p								13	17	23	25	Minimal + adaptive
Space Shuffle Ideal (S2-ideal)	p	4	4	4	4	4	4	4	8	8	8	8	Look-up table
String Figure (SF)	p	4	4	4	4	4	4	4	8	8	8	8	Look-up table + greediest + adaptive

Figure 4.7: Evaluated network topologies and configurations ("N" indicates unsupported network scale).

Table 4.2: Summary of network topology features and requirements.

Topology	**Requires High-Radix Routers**	**Router Port**	**Reconfigurable Network**
ODM	No	No	No
AFB	Yes	Yes	No
S2-ideal	No	No	No
SF	No	No	Yes

4.5 Traffic Patterns and Experimental Setup

4.5.1 RTL Simulation Framework

This thesis evaluates String Figure via RTL design using SystemVerilog [118] and PyMTL [87]. This thesis develops synthesizeable RTL models of each network topology, routing protocol, memory node, router configuration, and wire lengths. Table 4.1 describes the

Table 4.3: Description of network traffic patterns.

Traffic Pattern	Formula	Description
Uniform Random	`dest = randint(0, nports-1)`	Each node produces requests to a random destination node in the network.
Tornado	`dest = (src+nports/2) % nports`	Nodes send packets to a destination node halfway around the network.
Hotspot	`dest = const`	Each node produces requests to the same single destination node.
Opposite	`dest = nports - 1 - src`	Sends traffic to opposite side of network like a mirror.
Nearest Neighbor	`dest = src + 1`	Each node sends requests to its nearest "neighbor" node, one away.
Complement	`dest = src` $\oplus$ `(nports-1)`	Nodes send requests to their bitwise complement destination node.
Partition 2	`dest = randint(0,nports-1) & (nports/2-1) | (src & (nports/2))`	Partitions the network into two groups of nodes. Nodes randomly send within their group.

modeled system configurations. The experiments use the same configuration, timing, and energy parameters evaluated in previous works. The experiments estimate the dynamic energy used in the network using average picojoule-per-bit numbers to provide a fair comparison of memory access energy [68, 112, 144]. Network clock rate is the same as memory nodes clock speed, e.g., 312.5MHz with HMC-based memory nodes. Static energy is not evaluated, as static power saving is highly dependent on the underlying process management assumptions (e.g., race-to-idle). The experiments also model the network link latency based on wire length obtained from 2D grid placement of memory nodes. The configuration adds an extra one-hop latency with a wire length equal to ten memory nodes on the 2D grid (based on the wire length supported by HMC). The RTL simulator can run workload traces collected using an in-house trace generation tool, which is developed on top of Pin [90]. The experiments collect traces with 100,000 operations (e.g., grep for Spark-grep, queries for Redis) after workload initialization. The experiment trace generator models a cache hierarchy with 32KB L1, 2MB L2, and 32MB L3 with associativities of 4, 8, and 16, respectively. This trace generator does not contain a detailed core model and thus can only obtain the absolute instruction ID of each memory access.

Table 4.4: Description of evaluated real workloads.

Workload	Description
Spark-wordcount	A ”wordcount” job running on Spark, which counts the number of occurrences of each word in the Wikipedia data set provided in BigDataBench [139].
Spark-grep	A ”grep” job running on Spark, which extracts matching strings from text files and counts how many times they occurred with the Wikipedia data set provided in BigDataBench [139].
Spark-sort	A ”sort” job running on Spark, which sorts values by keys with the Wikipedia data set provided in BigDataBench [139].
Pagerank	A measure of Twitter influence. From the graph analysis benchmark in CloudSuite [42]. Twitter data set with 11M vertices.
Redis	An in-memory database system which simulates running 50 clients at the same time sending 100,000 total queries [6].
Memcached	From CloudSuite [42], which simulates the behavior of a Twitter caching server using the Twitter data set with 8 client threads, 200 TCP/IP connections, and a get/set ratio of 0.8.
Matrix Mul	Multiplying two large matrices stores in memory and storing their result in memory.
Kmeans	Clustering Algorithm partitions n observations into k clusters where each observation belongs to cluster with the nearest mean.

However, multiplying the instruction IDs by an average CPI number generates a timestamp for each memory access.

Considered topologies. The experiments compare String Figure to a variety of network topologies and routing protocols summarized in Figure 4.7. Table 4.2 describes their features and requirements. The number of router ports *does not include* the terminal port connecting to the local memory node. String Figure allows arbitrary network scale. However, to provide concrete examples in the evaluation and demonstrate support for elastic network scale, this thesis implements two basic topologies with 128 nodes (4 router ports) and 1296 nodes (8 router ports), respectively. The basic topologies are reconfigured to evaluate networks with fewer of nodes. Mesh is widely explored in previous memory network designs as one of offering the best performance among various topologies [67, 144]. This thesis implements a baseline Optimized Distributed Mesh (ODM) topology [67] for memory network. In addition, the experiments compare String Figure with several network designs optimized for scalability of distributed systems, including a 2D Adaptive Flattened Butterfly (AFB) [70] and S2 [143]. S2 does not

support down-scaling with the same original topology (it requires regenerating new topologies and routing tables with a smaller number of nodes). Therefore, the evaluation of S2 provides an impractical ideal baseline. The experiments refer to it as S2-ideal. Additionally, the experiment results focuses on evaluating performance, energy, and scalability of the network designs. However, most of the baseline topologies require high-radix routers [70, 81] and the number of ports and links continues to grow with network scale, leading to non-linear growth of router and link overhead in memory networks. Furthermore, none of the baseline topologies offer the flexibility and reconfigurability in memory networks as provided by String Figure.

Bisection bandwidth. To provide a fair point of comparison, this thesis evaluates network designs based on the same (or similar) bisection bandwidth. Because String Figure and S2 [143] have random network topologies, their empirical minimum bisection bandwidths are calculated by randomly splitting the memory nodes in the network into two partitions and calculating the maximum flow between the two partitions. The minimum bisection bandwidth of a topology is calculated from 50 random partitions. The experiments adopt the average bisection bandwidth across 20 different generated topologies. With a fixed network size, the bisection bandwidth of FB with high-radix routers can be much higher than the other topologies. However, mesh is lower. To match the bisection bandwidth, this chapter also evaluates an Adaptive FB (AFB) implemented by partitioned FB [21] with fewer links and an optimized DM (ODM) with increase the links per router to match the bisection bandwidth of String Figure and S2 at each memory network scale.

4.5.2 Workloads

This thesis evaluates both network traffic patterns and real workloads on the simulation framework. **Traffic patterns:** It evaluates different traffic patterns running on String Figure and baseline designs. Table 4.3 lists details about these traffic patterns. These traffic patterns are used to evaluate the memory network designs and expose the performance and scalability trends. The experiments sweep through various memory network sizes, router configurations, and routing protocols listed in Figure 4.7. To exercise the memory network design, each memory node sends requests (similar to attaching a processor to each memory node) at various injection rates. For example, given an injection rate of 0.6, nodes randomly inject packets 60% of the time. **Real workloads:** This thesis also evaluates various real workloads

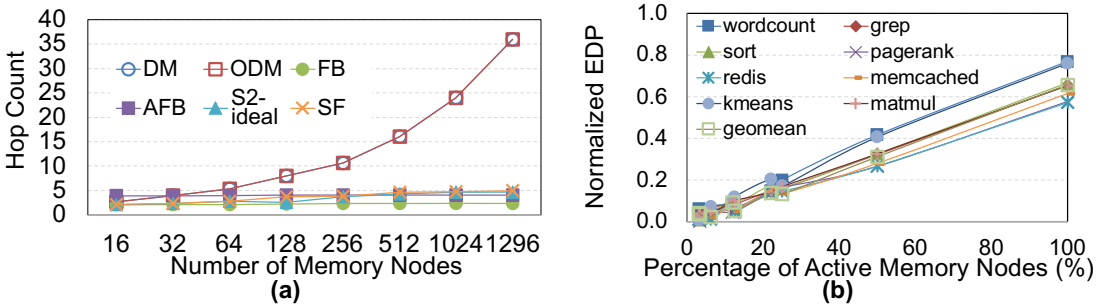


Figure 4.8: (a) Average hop counts of various network designs as the number of memory nodes increases. (b) Normalized energy-delay product (EDP) (the lower the better) with various workloads, when a certain amount of memory nodes are power gated off.

with trace-driven simulation. The experiments run various in-memory computing workloads in CloudSuite [42], BigDataBench [139], and Redis benchmark [6] on a Dell PowerEdge T630 server. Spark 1.4.1 [7] is used for all of the Spark jobs. Table 4.4 summarizes the workload and data set characteristics. The input data size of each real workload benchmark are scaled to fill the memory capacity. Data is distributed among the memory nodes based on their physical address.

4.6 String Figure Evaluation

This thesis evaluates String Figure with various metrics including average path lengths, network saturation, average packet latency, workload instruction throughput (IPC), memory dynamic energy, and energy efficiency. The evaluation shows that String Figure achieves close to or better than the performance and scalability of the best of prior designs (ODM, AFB, and S2-ideal), yet leads to lower energy consumption and higher energy efficiency with (i) fewer router ports and wires needed and (ii) elastic network scale.

Path lengths. Figure 4.8(a) shows the average shortest path lengths of various network designs across synthetic traffic patterns and real workloads. When the memory network has over 128 memory nodes, the average hop count of DM and ODM network increases superlinearly with increasing network size. Specifically, the average hop count of these two topologies is $\frac{2}{3}t$ where t is the average of their two dimensions. Rather, the other network topologies, S2-ideal, FB, AFB, and String Figure, do not incur significant increase in the average shortest path lengths

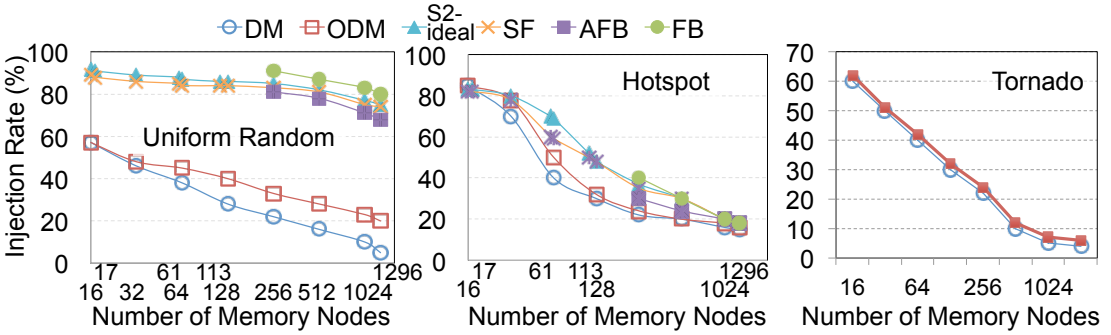


Figure 4.9: Network saturation points across various numbers of nodes.

in large-scale memory networks. FB achieves the best average shortest path lengths among all the network topologies, because it employs many more ports in routers than other topologies as the network scales up. With a maximum of eight ports per router, String Figure still achieves 4.75 and 4.96 average hop counts when the network scales up to 1024 and 1296 memory nodes, respectively. This thesis also evaluates 10% and 90% percentile shortest path lengths. String Figure can achieve 4 hops and 5 hops with over one thousand nodes, at 10% and 90% percentile, respectively. Therefore, String Figure path length is scalable to memory network size over one thousand nodes.

Network saturation. The experiments evaluate network saturation with several traffic patterns shown in Figure 4.9. String Figure can achieve close to the best of all other network architectures. In order to clearly visualize all the curves, only the results of the rest of the network architectures are shown. Traffic patterns `uniform random`, `hotspot`, and `tornado` are particularly noteworthy and show different results. The remaining traffic patterns `partition2`, `complement`, `opposite`, and `neighbor`, have similar behavior as shown. In almost all traffic patterns, the mesh network topologies, DM and ODM, saturates first at the lowest injection rate. Nearest-neighbor routing is the exception to this. SF perform worse with nearest-neighbor than ODM. This is because in mesh topologies, nodes are always one-hop away from their nearest neighboring node. Note, that the experiments generated nearest-neighbor network traffic using the router IDs rather than number of hops. Therefore, "neighboring" nodes in SF are not necessarily one hop away from each other which means this network has higher latency. However, an exception to mesh saturating first is in networks with very few nodes. At the fewest node configuration (i.e., 16 nodes), ODM slightly edges out SF. However, as the num-

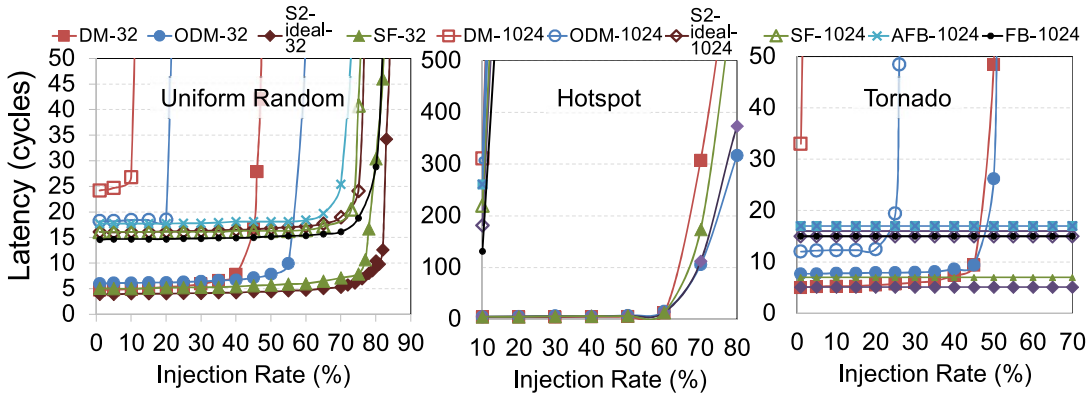


Figure 4.10: Performance of traffic patterns at less than one thousand nodes.

ber of memory nodes increases, SF scales significantly better. ODM also saturates at a higher injection rate than other network designs with `hotspot` traffic pattern. Network saturation in *tornado* traffic pattern are not observed with all topologies, except for mesh. Network latency remains steady even in high injection rates and large number of memory nodes. The reason is the geometric structure of the network designs. With either one of AFB, FB, S2-ideal, and SF, it is typically easy for packets in a network to traverse half or the entire network in just a hop or two to reach their destination. Traffic patterns, such as *tornado*, generate traffic in a mathematically geometric manner which is advantageous in such topologies.

Average packet latency. The experiments evaluate the average travel time (latency) between any two nodes in a network shown in Figure 4.10. Each traffic pattern graph shows the latency in the leftmost data point for each network. S2-ideal and SF appear to scale well with the number of memory nodes. As the number of nodes in the network increases, these topologies show almost no degradation in their network saturation points. SF has slightly longer latency than S2-ideal with networks down-scaled from the original size, because shortcuts and adaptive routing can degrade the randomness among network connections. However, SF still demonstrates lower latency than AFB at large network scales. The memory access latency of various traffic patterns are also evaluated.

Performance and energy of real workloads. The experiments evaluate system performance and memory dynamic energy consumption with several real workloads running in a memory system, where the total memory capacity is 8TB distributed across the 1024 (down-scaled from 1296) memory nodes in the network. It takes into account dynamic reconfiguration overhead

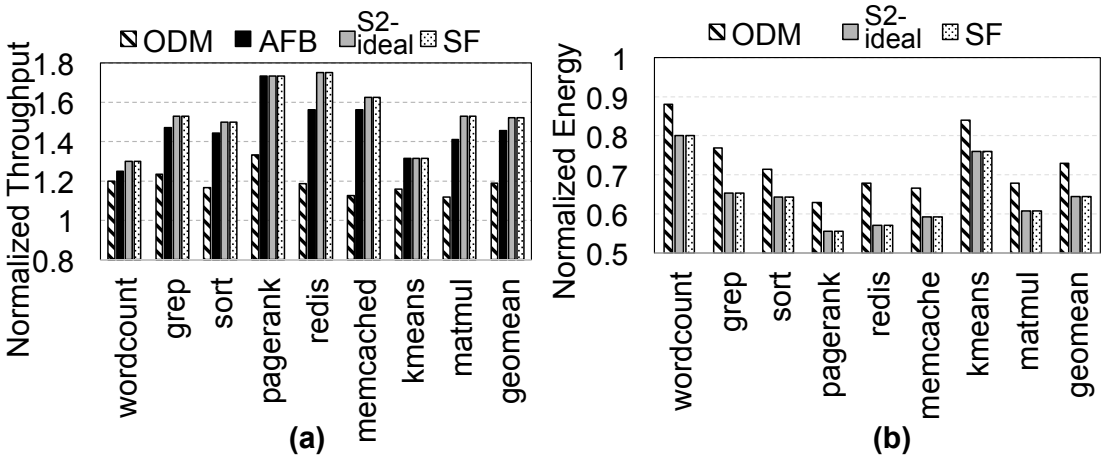


Figure 4.11: Normalized (a) system throughput (higher is better) and (b) dynamic memory energy (lower is better) with various real workloads.

to perform power gating in the RTL simulation by implementing SF reconfiguration mechanisms. The sleep and wake-up latency of a link is conservatively set to 680ns and 5μs similar to prior works [67, 144]. To minimize the performance impact of reconfiguration, the experiments set the reconfiguration granularity (i.e., the minimum allowed time interval between reconfigurations) to be 100us. Figure 4.11(a) shows the throughput of real workloads with varying memory network architectures, normalized to DM. The results demonstrate that String Figure can achieve close to the best performance across various workloads. The String Figure design achieves 1.3$\times$ throughput compared with ODM. Figure 4.11(b) illustrates normalized memory dynamic energy consumption with the experiment workloads, normalized to AFB. String Figure design can achieve the lowest energy consumption across these network topologies. S2-ideal also achieves similarly low energy consumption, due to its energy reduction in routing. On average, SF reduces energy consumption by 36% compared with AFB.

Memory network power management. The experiments also evaluate memory network power management by powering gating off various portions of the memory system with total 1296 nodes. Figure 4.8(b) shows the energy efficiency of String Figure's power management by considering both energy saving and system performance overhead. As demonstrated in the results, String Figure can achieve significantly improved energy efficiency, as more parts of the memory network are power gated.

Chapter 5

Related Work

This chapter discusses the related work on both persistent memory and memory networks. It also contains related work on prefetching, uncacheable logging, and RDMA networks.

5.1 Prefetching and Logging in Persistent Memory

Exploiting Memory Accesses through Prefetching. Simple prefetching provided a straightforward way to predict memory access patterns and improve cache performance. More complex adaptive prefetching techniques have sought to focus on certain applications and scenarios that commonly arise. In [117] and [23], the prefetching solutions focus on improving file I/O. In [49], prefetching is used for access patterns in file systems. Prefetching can be used in large workload systems with multiple streams [50] to adapt to changing workloads. All of these papers show the the success of the prefetcher used to address memory-specific problems and applications.

Persistent memory provides another subset of memory-related performance issues to be addressed, discussed in previous section. The prefetcher is an appropriate block for dealing this family of persistent applications because of its close ties to memory and memory accesses. EFetch similarly seeks to exploit what other works have used to justify changes to prefetching: the predictability of memory access patterns for those applications. In the case of persistent memory, the memory access patterns are predictable which justifies the eFetch mechanism.

Logging is a very common way to implement persistent memory [135] [30] [72]. Other works have developed techniques to improve upon logging and how it interacts with

the persistent data structures. Opting for redo logging over undo logging is one such technique [135] [30] that yield greater performance from relaxing writing back committed updates. EFetch relies on this precedent, but also analyzes the effect under undo logging systems. In [16], they propose an alternative logging scheme to the standard write-ahead logging. This improvement comes from shorter data recovery time and decreased memory footprint. EFetch can also work with this, but instead the value would be prefetched ahead of the log.

Similar to other persistent memory designs [136] [31] [134] [123], eFetch requires modification to legacy code to adapt eFetch for the persistent applications. However, Section 2.4 outlines a fairly lightweight software API that requires little change to the code.

Persistent memory work within Operating Systems and file systems [142] [38] shift the responsibility of logging data and persistence implementation to the systems level. They rely on existing structures and file system techniques. Although eFetch does this at the user-level, the software interface may also be implemented at the systems level and the hardware changes can also apply. A common thread between many of these logging schemes is that their designs have minimal to no hardware support. This thesis and other works have demonstrated benefit in coupling hardware support for persistent memory.

Other such studies have proposed hardware support for log-based persistent memory and are viable in improving system performance. Works like [89] and [73] have more hardware-intensive solutions and both show an improvement over baseline software-only systems. The eFetch hardware changes and accompanying software interface are lightweight (low complexity) and retain similar advantages.

Hardware Support for Persistent Memory without Prefetching. Prefetching isn't the only hardware block used for improving the performance of persistent memory applications. Prior works make changes to other hardware blocks to provide support and assistance [116]. EFetch is also not the first to propose or implement a solution that is complemented or supported by hardware [93] [33] [147] [98] [111] [65] [86] [110] [89] [149]. Most of these papers also use a software API similar to eFetch, but rely on other hardware blocks to support their design. EFetch exploits the predictability of the persistent memory applications to provide a lightweight solution. Efetch improves the performance under these systems while maintaining the performance in non-persistent systems at low cost in the hardware.

5.2 Logging and Persistent Memory

Compared to prior work in architecture support for persistent memory systems, the undo+redo logging design described in this thesis further relaxes ordering constraints on caches with less hardware cost.[1]

Hardware support for logging. Several recent studies proposed hardware support for log-based persistent memory design. Lu *et al.* proposes custom hardware logging mechanisms and multi-versioning caches to reduce intra- and inter-transaction dependencies [89]. However, they require both large-scale changes to the cache hierarchy and cache multi-versioning support. Kolli *et al.* proposes a delegated persist ordering [73] that substantially relaxes persistence ordering constraints by leveraging hardware support and cache coherence. However, the design relies on snoop-based coherence and a dedicated persistent memory controller. Instead, undo+redo logging is flexible because it directly leverages the information already in the baseline cache hierarchy. ATOM [63] and DudeTM [84] only implement either undo or redo logging. As a result, the studies do not provide the level of relaxed ordering offered by undo+redo logging described in this thesis. In addition, DudeTM [84] also relies on a *shadow* memory which can incur substantial memory access cost. Doshi *et al.* uses redo logging for data recoverability with a backend controller [37]. The backend controller reads log entries from the log in memory and updates data in-place. However, this design can unnecessarily saturate the memory read bandwidth needed for critical read operations. Also, it requires a separate victim cache to protect from dirty cache blocks. Instead, undo+redo hardware logging directly uses dirty cache bits to enforce persistence.

Hardware support for persistent memory. Recent studies also propose general hardware mechanisms for persistent memory with or without logging. Recent works propose that caches may be implemented in software [80], or an additional non-volatile cache integrated in the processor [74,146] to maintain persistence. However, doing so can double the memory footprint for persistent memory operations. Other works [85, 129, 148] optimize the memory controller to improve performance by distinguishing logging and data updates. Epoch barrier [30,32,109] is proposed to relax the ordering constraints of persistent memory by allowing coarse-grained transaction ordering. However, epoch barriers incur non-trivial overhead to the cache hierarchy. Furthermore, system performance can be sub-optimal with small epoch sizes, which is observed

[1] Volatile TM supports concurrency but does not guarantee persistence in memory.

in many persistent memory workloads [102]. Undo+redo hardware logging, however, design uses lightweight hardware changes on existing processor designs without expensive non-volatile on-chip transaction buffering components.

Persistent memory design in software. Mnemosyne [135] and REWIND [27] are prior works that utilize write-ahead logging implemented in software. These rely on instructions, such as `clflush`, `clwb`, and `pcommit`, to achieve persistency and enforce log-data ordering in their critical path. Undo+redo logging does not require these persistent instructions that have shown can clog the pipeline and are often inefficient or unnecessary. JUSTDO [60] logging also relies on these instructions (for manual flushing by the programmer). Additionally, these works' programming models all falter because Intel's `pcommit` instruction is now deprecated. Undo+redo logging works on legacy code and does not have strict functionality requirements on the programming model. This makes it immune to relying on instructions or software functions that become deprecated.

5.3 RDMA and Memory Networks

To the knowledge of this thesis, String Figure is the first memory network architecture that offers both scalability and elastic network scale in a single design. Most previous memory network designs do not take into account scalability as a primary design goal. Kim *et al.* [67, 68] explored memory network with mesh, butterfly, and dragonfly topologies with 64 HMC-based memory nodes. The study showed that distributed mesh outperforms other network topologies at this scale. Zhan *et al.* [144] investigated performance and energy optimization on a mesh-based memory network design up to 16 memory nodes. Poremba *et al.* [112] extended the memory network capacity to 2TB implemented by 128 HMC-like memory nodes used in CPU+GPU systems. However, the memory nodes are mapped separate processor channels, rather than shared by all the processors. Fujiki *et al.* [47] proposes a random network topology to support scalability of memory networks, yet does not support the flexibility and reconfigurability as String Figure.

Scalability and flexibility are central themes in data center network [69–71, 106, 122, 143]. Recent planar topologies, such as a Flattened Butterfly [70] and Dragonfly [69], offer promising scalability and high network throughput. However, these designs require high-radix routers and substantial increase of number of ports, which can impose non-linearly increasing

router area and power [81]. This leads to prohibitively high cost in routers and the amount wiring at large-scale memory network. Furthermore, butterfly-like topologies typically have symmetric layout. This can lead to isolated nodes or suboptimal routing, when subsets of nodes are turned down. Jellyfish [122] employs random topology to achieve close-to-optimal network throughput and incremental growth of network scale. Yet, Jellyfish provides high throughput by requiring k-shortest path routing; the size of forwarding tables per router can increase super-linearly with the number of routers in the network. This is impractical in a memory network, where routers have limited storage space. String Figure uses greedy routing due to the topology with randomly assigned coordinates in multiple spaces. As such, String Figure can achieve both high-throughput routing and constant forwarding state per router. S2 [143] adopts random topologies and computation-based routing mechanisms with scalable routing tables. Yet, S2 [143] requires cable plug in/out to increase the network size, which is impractical in memory networks that have pre-fabricated link wires. S2 does not support network downscaling, unlike String Figure.

Recent NoC designs tackle scalability and fault tolerance issues, when interconnecting processor cores. Slim NoC [21] enables low-diameter network in NoC. However, the design requires increasing router ports and wires as the network scales up. Furthermore, these topologies does not support the level of flexibility and reconfigurability as String Figure. Small world network [106] also employs greedy routing, but it does not produce the shortest paths and can be difficult to be extended to perform multi-path routing that can fully utilize network bandwidth. Previous network fault tolerance schemes [13, 18, 40, 41, 43, 77, 107, 113, 138] allow NoC to continue efficient functioning when routers are taken out of the network. However, most previous designs are developed for limited network scales and certain network topologies [40] and impose high router area overhead by employing one or several routing tables [13, 18, 41, 43, 77, 107, 113, 138].

Chapter 6

Conclusion

This thesis proposed three designs for improving the performance and reducing the hardware and energy cost of state-of-the-art memory systems. It also analyzed the balance between hardware and software design and examined the costs and trade-offs of each. The experimental evaluation of all three of these designs over a variety of benchmarks yielded higher performance with lower energy cost and lightweight hardware overhead. The rest of this chapter summarizes the key points and primary contributions of the this thesis and the proposed designs. The contributions of each design are outlined.

6.1 Thesis Summary and Contributions

This thesis proposed eFetch, a persistent-aware prefetcher design with two intelligent prefetching mechanisms. These mechanisms use prefetching to i) relax the ordering constraints between persistent memory access, and ii) parallelize logging and working data updates in redo and undo log based systems, respectively. EFetch addressed the issue of over-prefetching of log updates and leverages prefetching for relaxing ordering constraints in persistent memory applications.

Popular persistent memory techniques increasingly rely on the immediate availability and reuse of log-based and other persistence mechanisms. Therefore, hardware units that clash with the accessibility of these instruments manifest in the performance loss incurred from implementing persistent memory. The work in this chapter re-examines the design of a commonly-used hardware prefetcher amid expanding demand for persistent memory and nonvolatile tech-

nologies.

The experimental evaluation on diverse persistent workloads show that despite strictly less, yet smarter prefetching, there is a net improvement in throughput and performance. These gains outweigh the costs of less aggressive prefetching, but significant gains are made in terms of performance and dynamic power consumption. This holds true across multi-core implementation which underlines the complexity-effectiveness and scalability of eFetch.

The contributions of eFetch are as follows:

- This thesis identified new performance and energy issues associated with current hardware prefetching schemes in persistent memory systems. The design also identify the opportunities in leveraging prefetching to improve system throughput and reduce data movement across the memory bus by parallelizing logging and working data updates.
- This thesis proposed a new persistence-aware hardware prefetching scheme to improve persistent memory system throughput and energy-efficiency. This prefetching scheme consists of three design principles, including a persistent write tracking scheme that prevents prefetching of non-reusable log data, a redo log prefetching mechanism that accommodates working data updates and dependent reads with a single prefetching, and a undo logging triggered prefetching mechanism that proactively updates working data before undo logging is completed.
- This thesis presented a set of efficient design techniques to implement eFetch, including software interface, log address buffer design, prefetching policies, prefetching controller, and hardware support for memory barriers.

This thesis proposed an efficient hardware undo+redo logging design that relaxes the write-order control in persistent memory. It did this by capitalizing on existing caching policies. The design consists of two mechanisms. A **Hardware Logging (HWL)** unit automatically performs undo+redo logging by relying on write-back write-allocate caching, widely used in processors. Persistent memory writes trigger HWL to create a log for the data. The values required for a log entry: its address, old value, and new value are readily available in the cache hierarchy. Therefore, this design utilizes the cache block writes to update the log with word-size values. Hardware undo+redo logging also uses a cache **Force Write-Back (FWB)**

mechanism to force write-backs of cached persistent working data efficiently while guaranteeing persistency. It did all this without non-volatile buffers or caches in the processor. This thesis demonstrated that hardware undo+redo logging excels over traditional software logging for persistent memory. The evaluation shows that undo+redo hardware logging significantly increases performance while reducing memory traffic and energy. This thesis summarizes the contributions of hardware undo+redo logging as follows:

- This was the first design to exploit the combination of undo+redo logging to relax ordering constraints on caches and memory controllers in persistent memory systems. This design relaxes the ordering constraints in a way that undo logging, redo logging, or copy-on-write alone cannot.
- This design enabled efficient undo+redo logging for persistent memory systems in hardware, which imposes substantially more challenges than implementing either undo- or redo- logging alone.
- This design used a hardware-controlled cache force write-back mechanism, which significantly reduces the performance overhead of force write-backs by efficiently tuning the write-back frequency.
- This design was implemented with lightweight software support and processor modifications.

This thesis also proposed String Figure, a high-performance, scalable, and flexible memory network architecture. With String Figure, this thesis examined the critical scalability and flexibility challenges facing memory network architecture design in meeting the increasing memory capacity demand of future cloud server systems. String Figure offered numerous benefits towards practical use of memory network architecture in server systems, such as scaling up to over a thousand memory nodes with high network throughput and low path lengths, arbitrary number of memory nodes in the network, flexible network scale expansion and reduction, high energy efficiency, and low cost in routers. The design goals of String Figure were achieved and are summarized as follows:

- **Scalability.** String Figure supports over a thousand memory nodes shared by all CPU sockets in a server. It met the three-part scalability requirement. **Path lengths:** When the

network size (i.e., the number of memory nodes) grows, the routing path lengths grows sub-linearly. **Routing overhead:** The computation and storage overheads of routing decision-making are sublinear and independent of the network scale. **Link overhead:** The number of required router ports and links is also either sublinear and independent of the network size.

- **Arbitrary network scale.** String Figure maintains a high-throughput interconnect of an arbitrary number of memory nodes, without the shape balance limitation of traditional topologies, such as a mesh or tree.

- **Elastic network scale.** String Figure supports flexible expansion and reduction of the network size, in terms of the number of memory nodes. The elastic network scale allows effective power management by turning on and off routers and corresponding links. It also enables design and implementation reuse across various server system configurations.

This thesis explored a subset of a larger question about how computer architects can build systems that work well with the latest advancement in memory technology and techniques. On one hand, traditional models such as in networks on-chip, storage systems, and prefetching allows for some accommodation of these new techniques with sufficient software support. On the other hand, however, trends in computer architecture point towards hardware design, acceleration, and specialization for exacting the full potential of these modern systems. There is no one definitive correct answer for which is the best path to pursue. Regardless, this thesis made the case for how hardware design can achieve tremendous improvements in performance and efficiency. The best chance of success for computer architects might be to innovate across the entire computing stack and reap the most benefits where they manifest.

6.2 Future Work

Hardware design for persistent memory is promising, and as introduced in Section 3.6.1, has a lot of potential in future work with RISC-V. Some work has already been done towards this. The following proposals summarizes the ongoing and future work of persistent memory in RISC-V, depicted in Figure 6.1:

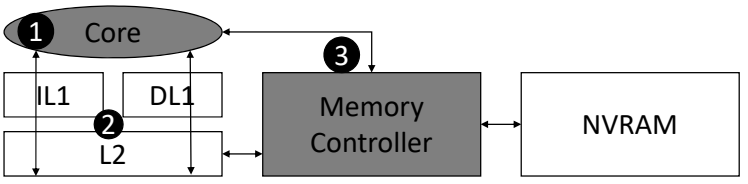


Figure 6.1: Overview of changes to RISC-V.

- Add persistent memory instructions to the RISC-V ISA. Most of this change is in the core (Figure 6.1❶).
- Create a RISC-V compatible version of pmemlib, required persistent memory evaluation using WHISPER.
- Modify the RISC-V architecture to enable persistent memory data paths (Figure 6.1❸), so that key techniques such as logging can be implemented in its hardware.
- Implement a state-of-the-art persistent memory design on a real RISC-V system and evaluate its performance and hardware costs.
- Demonstrate the overall effectiveness of the changes to RISC-V by running it on an FPGA using a real phase-change memory (PCM) device.

RISC-V Overview. There is a strong case for RISC-V, an open ISA [141]. Open ISAs were tried before RISC-V, but they never became popular. The demand for them is here now, thanks to the low cost IoTs and the desire for an open-source alternative to x86. RISC-V contains a standard set of optional extensions that have evolved over time. These extensions enable additional applications and usages for RISC-V, such as persistent memory.

ISA Extensions. A number of persistent memory instructions are needed in the RISC-V ISA. These instructions will serve as a baseline for future research in persistent memory. In particular, it is a strong basis for comparison with the ongoing RISC-V implementation of hardware undo+redo logging as described in this thesis. The following list defines these instructions:

- **ucst**: Instruction performs an uncacheable store to memory given the address and value. It completely bypasses the cache hierarchy and goes directly to the memory controller to be scheduled to memory.

- **ucld**: similar to `ucst`, but is an uncacheable load.
- **clwb**: Instruction forces a cache-line write-back to occur in the cache-line corresponding to its given address.
- **sfence**: Instruction orders stores in the program. All store instructions before this instruction must commit to memory before all store instructions following this instruction.

There are challenges with implementing these instructions in the current RISC-V instruction set. Specifically, the data path currently does not support uncacheable memory operations of any kind. Additionally, write-order control and non-standard cache operations are not supported either.

Architecture Changes. RISC-V requires specific architecture changes for the ISA. The aforementioned persistent memory instructions require updating the data path with new connections and I/O. The uncacheable load and store instructions must connect directly from the processor to the memory controller. Within the memory controller, RISC-V needs a write-combining buffer to coalesce the uncacheable memory requests to schedule them alongside in-flight cacheable requests. This functionality is already available in the design for hardware undo+redo logging. The `clwb` instruction requires changes to the RISC-V cache controllers to handle a new request for writing back cache lines. Aside from standard read or write requests, the new `clwb` function for the cache will cause a write-back of the dirty cache block associated with the given address without invalidating it.

The open-source RISC-V chip, Rocket (and its out-of-order version, BOOM) [26] was fully implemented in RTL and has already been taped out. They are synthesizable and programmable to FPGA boards. Ongoing work from this thesis has already modified their design to integrate the persistent memory techniques from hardware undo+redo logging. These changes can program on a Xilinx Zynq zc706 board. However, it has not yet been evaluated so completing this endeavor with a strong evaluation akin to those found in this thesis is future work. The added level of design with RISC-V provides programmers an aspect of control not obtainable through conventional commodity ISAs. By being able to directly modify the hardware of the machine while having the appropriate software support, it can provide a more thorough and robust study of persistent memory. It can also provide a level of accuracy in results not previously achieved, while exposing problems that can arise in the complexity of the

hardware.

These RISC-V changes will enable benchmarking of persistent memory applications and their execution, such as WHISPER [103]. Whisper is a persistent memory benchmark suite comprised of ten persistent memory applications. It covers a wide variety of persistent interfaces. It covers applications that access persistent memory directly, those using a persistent transactional library such as Mnemosyne [135], and those accessing persistent memory through a file system interface. Most recent persistent memory research, like in this thesis, evaluate using WHISPER because the community trusts its breadth and depth. Therefore, it is critical that RISC-V be able to run these benchmarks as well to properly support evaluating persistent memory benchmarks.

Persistent Memory Aware Systems. Even the most hardware-intensive persistent memory solutions require some amount of software or systems-level awareness. Specifically, crash consistency and recovery is important for proper functionality [109]. More robust systems-level work in persistent memory–persistent memory aware file systems such as NOVA [142] are dependent on the aforementioned persistent memory instructions. This ongoing and future work opens many such avenues of research into persistent memory systems thanks to the flexibility and adaptability of RISC-V architecture.

Bibliography

[1] Gen-Z Consortium. https://genzconsortium.org.

[2] Intel Xeon Phi processor 7200 family memory management optimizations. https://software.intel.com/en-us/articles/intel-xeon-phi-processor-7200-family-memory-management-optimizations.

[3] JEDEC publishes HBM2 specification as Samsung begins mass production of chips. https://www.anandtech.com/show/9969/jedec-publishes-hbm2-specification.

[4] Microsoft azure documentation. https://https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-memory#m-series.

[5] NVIDIA Tesla P100: Infinite compute power for the modern data center. http://www.nvidia.com/object/tesla-p100.html.

[6] Redis Benchmark. http://redis.io/topics/benchmarks.

[7] Spark 1.4.1. http://spark.apache.org/downloads.html.

[8] The Machine: A new kind of computer. https://www.labs.hpe.com/the-machine.

[9] ARM, ARMv8-a architecture evolution, 2016.

[10] Intel, a collection of linux persistent memory programming examples, https://github.com/pmem/linux-examples.

[11] Jung Ho Ahn, Sheng Li, O. Seongil, and Norman P. Jouppi. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *ISPASS*, pages 74–85. IEEE, 2013.

[12] Jung Ho Ahn, Sheng Li, O. Seongil, and N.P. Jouppi. Mcsima+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 74–85, 2013.

[13] Konstantinos Aisopos, Andrew DeOrio, Li-Shiuan Peh, and Valeria Bertacco. ARIADNE: Agnostic reconfiguration in a disconnected network environment. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 298–309, 2011.

[14] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, 2008.

[15] AMD. AMD Radeon R9 series graphics cards. http://www.amd.com/en-us/products/graphics/desktop/r9.

[16] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[17] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, pages 465–476, 2005.

[18] Marco Balboni, José Flich, and Davide Bertozzi. Synergistic use of multiple on-chip networks for ultra-low latency and scalable distributed routing reconfiguration. In *Proceedings of the 2015 Design, Automation and Test in Europe Conference*, pages 806–811, 2015.

[19] Jeff Barr. EC2 in-memory processing update: Instances with 4 to 16 TB of memory and scale-out SAP HANA to 34 TB. 2017.

[20] Jeff Barr. Now available – EC2 instances with 4 TB of memory. 2017.

[21] Maciej Besta, Syed Minhaj Hassan, Sudhakar Yalamanchili, Rachata Ausavarungnirun, Onur Mutlu, and Torsten Hoefler. Slim NoC: A low-diameter on-chip network topology for high energy efficiency and scalability. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 43–55, 2018.

[22] Timo Bingmann. STX B+ Tree, Sept. 2008, http://panthema.net/2007/stx-btree.

[23] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based i/o prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, 19(2):111–170, May 2001.

[24] Carlo Cagli. Characterization and modelling of electrode impact in HfO2-based RRAM. In *Proceedings of the Memory Workshop*, 2012.

[25] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 387–400, 2012.

[26] Christopher Celio, David A. Patterson, and Krste Asanovi. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.

[27] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.*, 8(5):497–508, January 2015.

[28] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. *SIGARCH Comput. Archit. News*, 22(2):223–232, April 1994.

[29] Neal Christiansen. Storage class memory support in the Windows OS. In *SNIA NVM Summit*, 2016.

[30] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-heaps: making persistent objects fast and safe with

next-generation, non-volatile memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[31] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News*, 39(1):105–118, March 2011.

[32] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[33] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, 2004.

[35] Kevin Deierling. Persistent memory over fabric. In *SNIA NVM Summit*, 2016.

[36] Cristian Diaconu. Microsoft SQL Hekaton c towards large scale use of PM for in-memory databases. In *SNIA NVM Summit*, 2016.

[37] K. Doshi, E. Giles, and P. Varman. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89, 2016.

[38] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, 2014.

[39] B. Falsafi and T.F. Wenisch. *A Primer on Hardware Prefetching*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2014.

[40] Mohammad Fattah, Antti Airola, Rachata Ausavarungnirun, Nima Mirzaei, Pasi Liljeberg, Juha Plosila, Siamak Mohammadi, Tapio Pahikkala, Onur Mutlu, and Hannu Tenhunen. A low-overhead, fully-distributed, guaranteed-delivery routing algorithm for faulty network-on-chips. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, pages 18:1–18:8, 2015.

[41] Chaochao Feng, Zhonghai Lu, Axel Jantsch, Minxuan Zhang, and Zuocheng Xing. Addressing transient and permanent faults in NoC with efficient fault-tolerant deflection router. In *IEEE TVLSI*, 2013.

[42] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *ACM SIGARCH Computer Architecture News*, 40(1):37–48, 2012.

[43] David Fick, Andrew DeOrio, Gregory Chen, Valeria Bertacco, Dennis Sylvester, and David Blaauw. A highly resilient routing algorithm for fault-tolerant NoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 21–26, 2009.

[44] Sam Fineberg and Pankaj Mehra. Log driven storage controller with network persistent memory. *U.S. Patent Number 8,706,687*, 2014.

[45] The Apache Software Foundation. Spark, 2014.

[46] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. Stride directed prefetching in scalar processors. *SIGMICRO Newsl.*, 23(1-2):102–110, December 1992.

[47] Daichi Fujiki, Hiroki Matsutani, Michihiro Koibuchi, and Hideharu Amano. Randomizing packet memory networks for low-latency processor-memory communication. In *Proceedings of Parallel, Distributed, and Network-Based Processing (PDP), Euromicro International Conference on*, 2016.

[48] Ilya Ganusov and Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proceedings of the 14th International Conference on Parallel*

Architectures and Compilation Techniques, PACT '05, pages 350–360, Washington, DC, USA, 2005. IEEE Computer Society.

[49] Javier Garcia Blas, Florin Isaila, Jesus Carretero, David Singh, and Felix Garcia-Carballeira. Implementation and evaluation of file write-back and prefetching for mpi-io over gpfs. *Int. J. High Perform. Comput. Appl.*, 24(1):78–92, February 2010.

[50] Binny S. Gill and Luis Angel D. Bathen. Amp: Adaptive multi-stream prefetching in a shared cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 26–26, Berkeley, CA, USA, 2007. USENIX Association.

[51] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[52] Swapnil Haria, Sanketh Nalli, Michael M. Swift, Mark D. Hill, Haris Volos, and Kimberly Keeton. Hands-off persistence system (HOPS). In *Non-volatile Memories Workshop*, 2017.

[53] S. Hassan and S. Yalamanchili. Bubble Sharing: Area and energy efficient adaptive routers using centralized buffers. In *Proceedings of the NOCS*, 2014.

[54] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[55] Xing Hu, Matheus A. Ogleari, Jishen Zhao, Shuangchen Li, Abanti Basak, and Yuan Xie. Persistence parallelism optimization: A holistic approach from memory bus to rdma network. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 494–506, 2018.

[56] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. Nvram-aware logging in transaction systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[57] Intel. Intel architecture instruction set extensions programming reference, 2016. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[58] Intel. Intel persistent memory programming, 2016. http://pmem.io.

[59] Intel and Micron. Intel and Micron produce breakthrough memory technology, 2015. http://newsroom.intel.com/community/intel_newsroom/.

[60] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *SIGPLAN Not.*, 51(4):427–442, March 2016.

[61] Animesh Jain, Ritesh Parikh, and Valeria Bertacco. High-radix on-chip networks with low-radix routers. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 289–294, 2014.

[62] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, 1997.

[63] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Feb 2017.

[64] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 1–12, 2014.

[65] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. Reducing the cost of persistence for nonvolatile heaps in end user devices. In *20th IEEE International Symposium On High Performance Computer Architecture (HPCA)*. IEEE, 2014.

[66] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 380–392, 2016.

[67] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. Memory-centric system interconnect design with hybrid memory cubes. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, pages 145–156, 2013.

[68] Gwangsun Kim, Minseok Lee, Jiyun Jeong, and John Kim. Multi-gpu system design with memory networks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 484–495, 2014.

[69] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable Dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 77–88, 2008.

[70] John Kim, William J. Dally, and Dennis Abts. Flattened Butterfly: A cost-efficient topology for high-radix networks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 126–137, 2007.

[71] Michihiro Koibuchi, Hiroki Matsutani, Hideharu Amano, D. Frank Hsu, and Henri Casanova. A case for random shortcut topologies for HPC interconnects. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 177–188, 2012.

[72] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2016.

[73] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th International Symposium on Microarchitecture*, pages 1–13, 2016.

[74] Chun-Hao Lai, Jishen Zhao, and Chia-Lin Yang. Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 5:1–5:6, New York, NY, USA, 2017. ACM.

[75] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977.

[76] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change

memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture*, pages 2–13, 2009.

[77] Doowon Lee, Ritesh Parikh, and Valeria Bertacco. Brisk and limited-impact NoC routing reconfiguration. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 306:1–306:6, 2014.

[78] Jaekyu Lee, Si Li, Hyesoon Kim, and Sudhakar Yalamanchili. Adaptive virtual channel partitioning for network-on-chip in heterogeneous architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 18(4):48:1–48:28, October 2013.

[79] Mengjie Li, Matheus A. Ogleari, and Jishen Zhao. Logging in persistent memory: to cache, or not to cache? In *Proceedings of the International Symposium on Memory Systems*, pages 1–3, 2017.

[80] P. Li, D. R. Chakrabarti, C. Ding, and L. Yuan. Adaptive software caching for efficient nvram data persistence. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 112–122, May 2017.

[81] Shang Li, Po-Chun Huang, David Banks, Max DePalma, Ahmed Elshaarany, Scott Hemmert, Arun Rodrigues, Emily Ruppel, Yitian Wang, Jim Ang, and Bruce Jacob. Low latency, high bisection-bandwidth networks for exascale memory systems. In *Proceedings of the Second International Symposium on Memory Systems*, pages 62–73, 2016.

[82] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[83] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 267–278, 2009.

[84] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for

persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. ACM.

[85] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified working memory and persistent store architecture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–470, 2014.

[86] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. Nvm duet: Unified working memory and persistent store architecture. *SIGARCH Comput. Archit. News*, 42(1):455–470, February 2014.

[87] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A unified framework for vertically integrated computer architecture research. In *47th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 280–292, 2014.

[88] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence in transactional persistent memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–13, 2015.

[89] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *ICCD*, 2014.

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

[91] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Transcendent memory and linux. In *Proceedings of the Linux Symposium*, pages 191–200, 2009.

[92] Aline Mello, Leonel Tedesco, Ney Calazans, and Fernando Moraes. Virtual channels in networks on chip: Implementation and evaluation on hermes NoC. In *Proceedings of*

the 18th Annual Symposium on Integrated Circuits and System Design, pages 178–183, 2005.

[93] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. A case for efficient hardware/software cooperative management of storage and memory. In *Proceedings of the 5th Workshop on Energy-Efficient Design (WEED), held in conjunction with the International Symposium on Computer Architecture (ISCA-40)*, 2013.

[94] Micron. Hybrid memory cube specification 2.1.

[95] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[96] Gordon E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. 2000.

[97] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: log-based transactional memory. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pages 1–12, 2006.

[98] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Nathan Binkert, and Parthasarathy Ranganathan. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the ACM Conference on Timely Results in Operating Systems*, pages 1–17, 2013.

[99] Jeff Moyer. Persistent memory in Linux. In *SNIA NVM Summit*, 2016.

[100] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *Supercomput. Front. Innov.: Int. J.*, 1(3):19–55, October 2014.

[101] Lifeng Nai and Hyesoon Kim. Instruction offloading with HMC 2.0 standard: A case study for graph traversals. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 258–261, 2015.

[102] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of*

th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 1–14, 2017.

[103] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, 2017.

[104] Matheus A. Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but no force: Efficient hardware-based undo+redo logging for persistent memory systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–14, 2018.

[105] Matheus A. Ogleari, Ye Yu, Chen Qian, Ethan L. Miller, and Jishen Zhao. String figure: A scalable and elastic memory network architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–14, 2019.

[106] Umit Y. Ogras and Radu Marculescu. "it's a small world after all": NoC performance optimization via long-range link insertion. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(7):693–706, July 2006.

[107] Ritesh Parikh and Valeria Bertacco. uDIREC: Unified diagnosis and reconfiguration for frugal bypass of NoC faults. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 148–159, 2013.

[108] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[109] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceedings of the International Symposium on Computer Architecture*, pages 1–12, 2014.

[110] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. *SIGARCH Comput. Archit. News*, 42(3):265–276, June 2014.

[111] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7(2), 2013.

[112] Matthew Poremba, Itir Akgun, Jieming Yin, Onur Kayiran, Yuan Xie, and Gabriel H. Loh. There and back again: Optimizing the interconnect in networks of memory cubes. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 678–690, 2017.

[113] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 198–209, 2004.

[114] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, 2009.

[115] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[116] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pages 1–13, 2015.

[117] Dan Revel, Dylan McNamee, David Steere, and Jonathan Walpole. Adaptive prefetching for device independent file i/o. Technical report, 1997.

[118] David I. Rich. The evolution of systemverilog. *IEEE Des. Test*, 20(04):82–84, July 2003.

[119] Corey Sanders. Announcing 4 TB for SAP HANA, single-instance SLA and hybrid use benefit images. 2016.

[120] SAP. SAP HANA: an in-memory, column-oriented, relational database management system, 2014.

[121] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[122] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 17–17, 2012.

[123] SNIA. NVM programming model (NPM), version 1, 2013. http://snia.org/sites/default/files/NVMProgrammingModel_v1.pdf.

[124] V. Sousa. Phase change materials engineering for RESET current reduction. In *Proceedings of the Memory Workshop*, 2012.

[125] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, 2007.

[126] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 63–74, Washington, DC, USA, 2007. IEEE Computer Society.

[127] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, 2001.

[128] Matthias Bo Stuart, Mikkel Bystrup Stensgaard, and Jens Sparsø. The renoc reconfigurable network-on-chip: Architecture, configuration algorithms, and evaluation. *ACM Trans. Embed. Comput. Syst.*, 10(4):45:1–45:26, November 2011.

[129] Long Sun, Youyou Lu, and Jiwu Shu. DP2: Reducing transaction overhead with differential and dual persistency in persistent memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pages 24:1–24:8, 2015.

[130] Zehra Sura, Arpith Jacob, Tong Chen, Bryan Rosenburg, Olivier Sallenave, Carlo Bertolli, Samuel Antao, Jose Brunheroto, Yoonho Park, Kevin O'Brien, and Ravi Nair. Data access optimization in a processing-in-memory system. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pages 6:1–6:8, 2015.

[131] Christian Szegedy, Wei Liu, and Yangqing Jia et al. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[132] The Next Platform. Baidu eyes deep learning strategy in wake of new GPU options. In *www.nextplatform.com*, 2016.

[133] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000.

[134] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[135] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[136] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. *SIGARCH Comput. Archit. News*, 39(1):91–104, March 2011.

[137] VoltDB. Voltdb: Smart data fast, 2014.

[138] Eduardo Wachter, Augusto Erichsen, Alexandre Amory, and Fernando Moraes. Topology-agnostic fault-tolerant NoC routing method. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1595–1600, 2013.

[139] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, K. Zhan, Xiaona

Li, and Bizhu Qiu. Bigdatabench: a big data benchmark suite from internet services. In *HPCA*, pages 488–499. IEEE, 2014.

[140] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[141] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovi. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[142] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[143] Ye Yu and Chen Qian. Space shuffle: A scalable, flexible, and high-bandwidth data center network. In *Proceedings of the 2014 IEEE 22Nd International Conference on Network Protocols*, ICNP '14, pages 13–24, 2014.

[144] Jia Zhan, Itir Akgun, Jishen Zhao, Al Davis, Paolo Faraboschi, Yuangang Wang, and Yuan Xie. A unified memory network architecture for in-memory computing in commodity servers. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[145] Jishen Zhao, Sheng Li, Jichuan Chang, John L. Byrne, Laura L. Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi. Buri: Scaling big memory computing with transparent memory expansion. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

[146] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, pages 421–432, 2013.

[147] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Pro-*

ceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[148] Jishen Zhao, Onur Mutlu, and Yuan Xie. FIRM: Fair and high-performance memory control for persistent memory systems. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO-47)*, 2014.

[149] Jishen Zhao, Onur Mutlu, and Yuan Xie. FIRM: fair and high-performance memory control for persistent memory systems. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 153–165, 2014.

[150] W. Zhao, E. Belhaire, Q. Mistral, C. Chappert, V. Javerliac, B. Dieny, and E. Nicolle. Macro-model of spin-transfer torque based magnetic tunnel junction device for hybrid magnetic-CMOS design. In *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, pages 40–43, Sept 2006.

[151] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 14–23, New York, NY, USA, 2009. ACM.

[152] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techniques for mpeg benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782–796, Aug 2000.