UNIVERSITY OF CALIFORNIA SAN DIEGO

Building Scalable Architectures Using Emerging Memory Technologies

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

in

Computer Science (Computer Engineering)

by

Kunal Kishore Korgaonkar

Committee in charge:

Professor Steven Swanson, Chair
Professor Chung-Kuan Cheng
Professor Pamela Cosman
Professor Rajesh Gupta
Professor Ryan Kastner

2019

The Dissertation of Kunal Kishore Korgaonkar is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2019

DEDICATION

To all life on earth and special mention of them who gave me mine

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to acknowledge Professor Steven Swanson for his support as my advisor and the chair of my committee. I am grateful to him for guiding me through research and life. I would like to thank my committee members for the useful feedback I received from them concerning this thesis. I also want to thank all my lab members, collaborators, mentors, managers and colleagues for being part of my journey. I would also like to thank my funding agencies and industry sponsors for supporting me for my education. Lastly but not least, I would like to thank my family for their unconditional support.

Chapter 2, in most parts, is a reprint of the material as it is accepted for publication in the Proceedings of the *38th Annual ACM Symposium on Principles of Distributed Computing (PODC '19)*; Vorpal: Vector Clock-Inspired Ordering For Large Persistent Memory Systems; Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao and Steven Swanson, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in most parts, has been published in the Proceedings of the *45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 315-327; Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache; Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young and Hong Wang, 2018. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part is currently being reviewed for publication in the *IEEE Computer Architecture Letters (CAL '19)*; Bitlet Model: Understanding the Sweet Spots and Limits of Processing in Memory; Kunal Korgaonkar, Ronny Ronen, Anupam Chattopadhyay and Shahar Kvatinsky, 2019. The dissertation author was the primary investigator and author of this material.

VITA

2005	Bachelor In Engineering, Goa University, India
2010	Masters in Science, Indian Institute of Technology Madras, India
2019	Doctor of Philosophy, University of California San Diego, USA

PUBLICATIONS

Vorpal: Vector Clock-Inspired Ordering For Large Persistent Memory Systems; K. Korgaonkar, J. Izraelevitz, J. Zhao, S. Swanson; Principles of Distributed Computing (PODC), 2019 *(conditional accept)*

Bitlet Model: Understanding the Sweet Spots and Limits of Processing in Memory; K. Korgaonkar, R. Ronen, A. Chattopadhyay, S. Kvatinsky; IEEE Computer Architecture Letters (CAL), 2019 *(under review)*

Density Tradeoffs of NVM as a Replacement for SRAM LLCs; K. Korgaonkar, I. Bhati, H. Liu, J. Gaur, S. Manipatruni, S. Subramoney, T. Karnik, S. Swanson, I. A. Young, H. Wang; International Symposium on Computer Architecture (ISCA), 2018

Characterization of User's Behavior Variations for Design of Replayable Mobile Workloads; S. Patil, Y. Kim, K. Korgaonkar, I. Awwal, T. Rosing; Mobile Computing Applications and Services (MOBICASE), 2015

Implications of Shared-Data Synchronization Techniques on Multi-Core Energy Efficiency; A. Gautham, K. Korgaonkar, Patanjali S., S. Balachandran and V. Kamakoti; Power Aware Computing and Systems (HotPower), 2012

Size-proportional signature sharing for transactional memory systems; K. Korgaonkar, G. Kashyap and V. Kamakoti; Future Architectural Support for Parallel Programming (FASPP) 2012

Reconstructing hardware transactional memory for workload optimized systems; K. Korgaonkar, P. Jain, D. Tomar, K. Garimella and V. Kamakoti; Advanced Parallel Processing Technologies (APPT), 2011

Thread synchronization: from mutual exclusion to transactional memory; K. Korgaonkar and V. Kamakoti, IETE Technical Review, 2011.

HTM Design Spaces: Complete Decoupling from Caches (short paper); K. Korgaonkar, G. Kurian, M. Gautam and V. Kamakoti; ACM SIGOPS Operating Systems Review (OSR), 2009

ABSTRACT OF THE DISSERTATION

Building Scalable Architectures Using Emerging Memory Technologies

by

Kunal Kishore Korgaonkar

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2019

Professor Steven Swanson, Chair

A confluence of trends is reshaping computing today. On one end, the massive amounts of data being generated by the proliferation of sensing and internet services are creating a demand for better computer architectures and systems. The other stream of the confluence is the nanotechnology advances that are unearthing new memory device technologies with the potential to replace (or be combined with) conventional memories.

Given these trends, this thesis examines emerging memory device technologies that provide a unique opportunity to build computer architectures with efficient and scalable data storage and processing capabilities. The associated memory architectures of these new systems promise to offer distinctive features such as intrinsic non-volatility, highly dense memory structures,

extremely low-power consumption and even embedded processing capabilities. Among others, some examples of emerging memory technologies with such features are PCM, 3D Xpoint, STT-RAM and ReRAM.

A central question with the new memory architectures built with emerging memory technologies is whether or not the resultant systems are scalable. Towards answering this question, this thesis identifies that conventional memory architecture specific scaling methods may not directly apply in case of emerging memory technologies. These methods were developed mostly for SRAM and DRAM, and today, they do not provide the desired outcomes for emerging memory technologies. As a result, there exist fundamental unsolved problems concerning scalability in building memory architectures. Unfortunately, this means that even though emerging memory technologies provide distinctive features, they may be largely left untapped.

Given the scalability concerns, this thesis then advocates a scalability-first approach for building computer architectures using emerging memory technologies while being aware of the limitations and opportunities associated with them. As demonstrations of the scalability-first approach, the thesis discusses several scalability problems encountered in systems using emerging memory technologies. It also brings out potential solutions for each of these problems in the form of novel techniques and tools.

For instance, the thesis discusses the problem and a solution for scaling write order enforcement mechanisms for data persistence on large non-volatile main memory systems, followed by the problem and a potential solution for scaling write bandwidth and thereby reducing memory interference on systems with dense non-volatile memory caches. Also discussed are methods for scaling system architectures with in-memory processing capability subject to its operational complexity and other limits.

The proposed scalability-first approach points to prospects and ways for better adoption of emerging memory technologies within existing systems. The approach and the solutions also lead to likely transition paths to even more scalable and markedly different systems of the future.

Chapter 1

Introduction

On the promise of scalable architectures using emerging memory technologies

The thesis is set in the context of the recent confluence of nanotechnology and architecture trends which provide a unique opportunity in realizing truly data-centric computer systems.

This introductory text discusses the emergence of new memory technologies and the features that these memories offer. Their unique features and the associated advantages bring about the need to build computer architectures that can best utilize them. Unfortunately, past scaling methods developed for conventional memory technologies may not be directly applicable in the case of new memories.

This chapter brings into focus the difficulties of realizing the full potential of emerging memories. The text introduces some concrete examples of scalability problems to demonstrate the nature of these difficulties. These scalability problems are described in details later through the course of this thesis.

As a path towards a solution, this chapter also introduces the proposed scalability-first approach for building systems using emerging memory technologies. It advocates that such an approach is essential for best utilizing the new memory technological trends. The chapter ends with an outline for the rest of the thesis.

1.1 The Confluence

A confluence of trends occurring today can potentially reshape computing forever. At one end, is the nanotechnology revolution leading to the discovery of new memory and storage device technologies. These devices are extending our understanding of materials and device physics at the tiniest scales. The other stream of the confluence comes from the proliferation of sensing and online services generating massive amounts of data, popularly known as the big-data revolution. The proliferation has set a stage for a closer examination of the inefficiencies of existing computer systems built with conventional memory and storage technologies.

The following paragraphs provide the reader with an introduction to nanotechnology trends. An introduction to the architecture trends follows. Each of these trends and their co-existence is a recurring theme in the rest of this thesis. The subsequent chapters of the thesis shall elaborate more on the specifics of certain memory technologies, as deemed necessary.

1.1.1 The Emergence of New Memory Technologies

Current trends in nanotechnologies are leading to new findings in emerging memory and storage device technologies. Several types of devices with varying properties such as non-volatility, density and in-memory computing capabilities are being investigated. In the past, the range of memory and storage options were limited (in terms of speed and cost-per-bit). More options are now or will be made available to system designers in the form of emerging memory technologies.

As an example, Intel's 3D-XPoint or Optane main memory was released recently and is now available for sale as real computer memory product. It provides cost-effective capacities (similar to FLASH storage media), as well as high speeds (closer to DRAM-based main memory). Over time, many of the new devices will potentially replace, or be combined with existing memory and storage technologies and thus will find their respective places in widely used computer systems.

This thesis shall refer to emerging storage and memory technologies as emerging memory technologies, given their likely use as main memories or as caches placed closer to the processors.

1.1.2 Trends in Data-centric Architectures

The other key trend witnessed today is the massive amounts of data being generated by the proliferation of sensing and internet services. In this respect, especially relevant today is the need to store and process large amounts of data efficiently. Unfortunately, modern computer architectures were not designed for efficiently performing data storage and processing tasks.

The inefficiencies related to data processing are deep-rooted. For example, some of these inefficiencies are as a result of the limitations of the Von-Neumann model. At the same time, it is unclear if any particular computing model (whether Von-Neumann or otherwise) can address all these inefficiencies effectively. Already, conventional CMOS trends, such as the slowing Moore's law and the unsustainable power densities, are making today's computer architectures less attractive option. These trends if continues may make today's computer architectures further unattractive for performing vast amounts of data processing and storage tasks.

Instead of the options discussed above, this thesis focusses on understanding the scalability of systems that attempt to utilize the emerging memory technologies. The thesis attempts to answer whether or not the new memory technologies provide a path towards more scalable and efficient data-centric systems.

1.2 Scalable Architectures Using Emerging Memory Technologies: Promise or Peril?

Given the data revolution at one end and the emergence of new memory technologies at the other end, there is an urgency to actively build new systems and architectures that by their very design can exploit well the unique properties of emerging memory technologies.

Architectures and systems that best utilize the emerging memory technology trends for efficiency and scalability will be the ideal outcome of the confluence. The key question is

whether such systems can be built. To understand whether truly data-centric architecture can be designed and built (or not), this thesis poses the following pertinent question:

How to build computer architectures using emerging memory technologies that can lead to efficient and scalable data storage and processing systems of the future?.

The following text discusses three distinct scalability problems associated with three distinct feature of new memories. The old method for scalability (for conventional memories) and the associated issues are highlighted, along with the new or proposed scalable method. The chapter compares the traditional scaling method used in conventional memories with new methods employed for better scalability in emerging memories.

1.2.1 Scalability in Persistence

Enforcing persist ordering is necessary to exploit the non-volatility feature of emerging memories since systems will continue to contain some volatile parts inside which re-ordering can occur. The volatile parts could be SRAM caches, interconnects or memory controller buffers. Programmer semantics necessitates some form of crash consistency support. However, current solutions to ensure persistent ordering do not scale with high core, memory controller and socket count.

The thesis identifies this problem and thus establishes that merely exposing features of the new memories, in this case, mechanisms to support non-volatility, is not sufficient. Instead, ensuring scalability of those mechanisms is needed. The fact that scaling does not occur across core, memory controller and socket count with current solutions, also indicates the need for re-architecting across major subsystems to address this issue.

Below is a concise summary of the write ordering scalability problem.

- **Feature:** Non-volatility
- **Memory Technology:** PCM, 3D-Xpoint or ReRAM

- **Old Method:** Write to DRAM and log in slow media like SDD/HDD.
- **Scalability Issue:** Simple persistent write ordering mechanisms do not scale
- **Proposed Method:** Persisting in non-volatile main memories with distributed algorithms

1.2.2 Scalability in Caching

Write bandwidth is linked to memory's density. Memories with high density are likely to have lower write bandwidth due to intrinsic technological factors like yield and reliability constraints. These bandwidth characteristics raise questions on scalability.

Traditional methods to manage cache capacities, including those proposed recently for non-volatile memory caches, heavily relies on replacement policies. However, replacement alone is insufficient to provide the guarantees on memory access bandwidth allocations. Allocation guarantees are necessary for an individual core's performance scalability. Guarantees are also required for system scaling with additional core count.

The on-chip caches with emerging memories is a good case study to understand why the drop-in replacement approach may not always work for emerging memories. High-performance processors typically require high on-chip memory access bandwidth. While the slow writes to the caches eat up useful bandwidth, actually deteriorating performance and cancelling the advantage of higher capacities.

Here is a concise summary of the write bandwidth scalability problem in dense caches built with non-volatile memories.

- **Feature:** Density or capacity
- **Memory Technology:** STT-RAM or SOT-RAM
- **Old Method:** SRAM caches and replacement policies for capacity management
- **Scalability Issue:** Old replacement policies do not scale
- **Proposed Method:** STT-RAM caches with replacement and queuing polices combined

1.2.3 Scalability in In-Memory Computations

Current in-memory computing proposals mostly focus on exploiting the in-memory computing ability of new memories. For in-memory computations performed using stateful in-memory logic, there exist limits of operational complexity. In this context, operational complexity refers to the exact number of memory cycles required to complete a given operation of a specific size.

There exist a lack of general understanding of the scalability of in-memory processing elements relative to traditional CMOS execution units. Not considering the operational complexity may lead to actual scalability issues in the end systems.

In the context of this problem, it is also worth highlighting that, even though executing inside memories can be viewed as an instance of non Von-Neumann model, scalability issues appear nevertheless.

Below is a summary of the in-memory computing scalability limits.

- **Feature:** In-memory logic
- **Memory Technology:** ReRAM or STT-RAM
- **Old Method:** CMOS for logic near memory
- **Scalability Issue:** In-memory logic does not scale with complexity and data sizes
- **Proposed Method:** Analytical complexity calculations and being logic complexity-aware

1.3 A Scalability-first Approach

This thesis shows that a confluence in nanotechnologies and data-centric architectures is creating unique opportunities as well as potential risks in the adoption of new memory technologies. This thesis attempts to deepen our understanding of the resultant risks, in particular, the scalability-related risks.

By investigating three scalability issues as case studies, the thesis provides evidence for its claims on the new memory technology adoption risks. The thesis highlights the shortcomings of many of the recent new memory architectural proposals from the literature and their inability to access the risks. Based on a better understanding of the scalability issues, the thesis proposes solutions for each of them.

The identification of adoption risks, particularly the scalability related risks, and thereafter coming up with systematic solutions to address these risks, demonstrate a *scalability-first* approach. Below is a summary of the type of new memory technology adoption risks that may exist, stated in more general terms. The order of the stated risks roughly is in the order of how research has progressed in this area. The list is not exhaustive but should serve as useful guidelines.

1. *Drop-in replacement to exploit new memory features may not work.* Although new memory and storage technologies are promising in many respects, they in most cases cannot be a drop-in replacement for existing technologies. The drop-in replacement may not work well, either creating a severe performance or sometimes semantical issue.

2. *Exploiting new memory features do not translate to scalability.* In many cases, the initial solutions proposed for building non-volatile memory architectures may not as scalable as thought to be. Ability to exploit certain properties of new memories should not be confused with scalability. Exploiting a feature is easier without concern for scalability, and much harder with it.

3. *Whether Von-Neumann model or even otherwise, new scalability issues may appear.* Whether the computational model is Von-Neumann or otherwise, the adoption of new memory technologies may require rearchitecting the software and hardware subsystems and the related interfaces of current systems. Even non Von-Neumann models may also face scalability issues. Perhaps they should not be seen as a panacea for scalability.

The three features or capabilities, namely, non-volatility, high-density and in-memory computing capability, are central and unique to the nature of new memories. The solutions proposed in this thesis in relation to these features, therefore, address very fundamental and general aspects in the design of new computer and memory architectures. With more proliferation of emerging memory technologies, the significance of these solutions will only increase with time.

1.4 Organization of the Thesis

The thesis describes the three scalability problems and the proposed scalable solutions in *Chapter 2*, *Chapter 3* and *Chapter 4*, respectively. The three chapters are concerned with scalability problems in relation to *non-volatility*, *high-density* and *in-memory computing capability* of emerging memories, respectively. Each of these three chapters serve to illustrate the scalability-first approach.

1.5 Acknowledgement

This chapter contains material from the Proceedings of the *38th Annual ACM Symposium on Principles of Distributed Computing (PODC '19)*; Vorpal: Vector Clock-Inspired Ordering For Large Persistent Memory Systems; Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao and Steven Swanson, 2019. The dissertation author was the primary investigator and author of this publication. The chapter also contains material from the Proceedings of the *45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 315-327; Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache; Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young and Hong Wang, 2018. The dissertation author was the primary investigator and author of this paper. Lastly, some of the content in this chapter is based on the *IEEE Computer Architecture Letters (CAL '19)*; Bitlet

Model: Understanding the Sweet Spots and Limits of Processing in Memory; Kunal Korgaonkar, Ronny Ronen, Anupam Chattopadhyay and Shahar Kvatinsky, 2019. The dissertation author was the primary investigator and author of this material.

Chapter 2

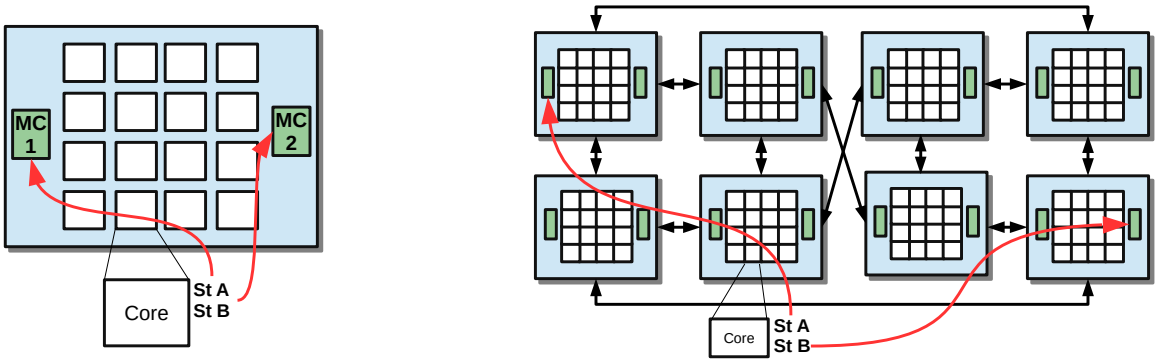
Scalability in Persistence

Scaling persistent ordering with more cores, memory-controllers and sockets

Non-volatile main memories (NVMMs), such as phase-change memory (PCM) [118], resistive RAM (ReRAM) [91], and 3D Xpoint [48], are likely to bring profound changes to many aspects of computer systems. While NVMMs offer promising opportunities, they also require systems using them to carefully order data updates into persistence in order to preserve consistency on system failures. A growing body of research [94, 22, 87, 53, 60, 62, 61, 80, 100] has been exploring hardware primitives to enforce these ordering constraints.

However, scalability of ordering support still remains an issue in NVMMs. Most previous work only considers ordering of NVMM updates through one or two memory controllers [94, 5, 22, 53, 60, 62, 61, 80]. But modern servers can include hundreds of cores and many memory controllers scattered across multiple sockets. In such a system, enforcing ordering constraints between stores generated at distributed cores and written into persistence (that is, *persisted*) at distributed memory controllers requires an efficient algorithm to capture and enforce these constraints.

Existing industry solutions [94, 5] address the problem by having a single core gather all the relevant cache lines together in one place followed by a series of cacheline flushes. The core essentially serves as the serializer slowing down the enforcement. Known research work [22, 53, 60, 62, 61, 80] uses explicit enqueue and acknowledgement (ack) messages of some form, between the CPU and memory controller (MC). For large multi-socket systems, where the MCs may be remotely placed on another socket, such schemes are especially problematic to implement



(a) Persistent store ordering problem on single socket server

(b) Persistent store ordering problem on eight socket NUMA server

Figure 2.1. This figure shows the expected memory system layout of a scale-up server with NVMM. In the figure parts (a) and (b) highlight the persistent store ordering problem in a distributed large-scale system setting.

and are non-scalable.

In this paper, we address these issues with a set of novel distributed algorithms for ordering updates to persistent memory. Our collection of algorithms, which we call Vector ORdered Persistence ALgorithm (Vorpal), uses a vector clock [79, 28] inspired representation to (1) construct a partial order over the stores executed in a distributed computer system and (2) efficiently enforce the order at distributed memory controllers.

In particular, this paper makes the following contributions:

- We describe a basic vector clock-inspired algorithm for persistence and demonstrate that it can implement the acquire-release persistent memory model.
- We propose new vector clock-inspired algorithms to address scalability problems of vector clocks as an ordering mechanism in many-core, multi-socket systems with a large number of memory controllers.
- We provide safety and liveness proofs for each of the proposed algorithms.

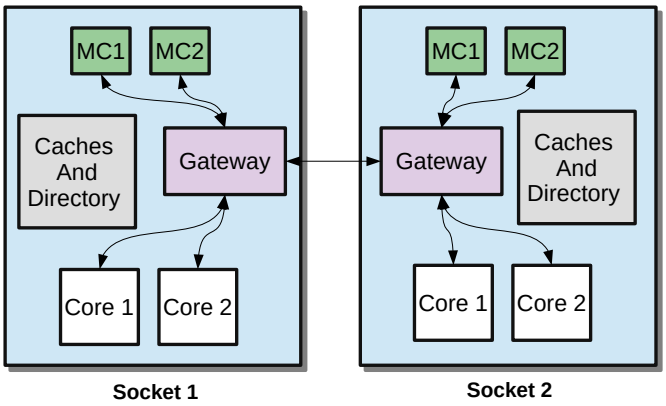


Figure 2.2. This figure shows the participating entities of Vorpal (for brevity we show just two cores and two controllers per socket)

2.1 Background

This section describes the necessary architectural and theory background for situating the Vorpal algorithms.

2.1.1 Acquire-Release Persistence (ARP)

In persistent memory programming, not only must we specify the happens-before ordering (which we write as $\prec_{HB}$ or $\prec$), we also must describe the persists-before ordering, that is, the ordering in which writes become persistent. The persist-before ordering (which we write as $\lessdot_{PB}$ or $\lessdot$) is controlled using a *persistency model*, which is specified on top of and along with the traditional memory consistency model. We chose a relatively relaxed persistency model called *acquire-release persistency* (ARP) [61] in order to demonstrate the flexibility of the Vorpal algorithms — trivially, Vorpal supports stricter persistency models.

As the name suggests, ARP is a persistency model built on top of release consistency. The ARP model is a refinement of traditional release consistency in that all happens-before orderings in the original consistency model are also persists-before orderings for all stores to persistent memory addresses (synchronization edges between store-release and load-acquire

also order the persistent stores of the synchronizing threads). In addition to the happens-before induced persistence orderings, ARP adds local *persist fences* that restrict the persist-before order within the local thread without incurring any happens-before edges. These persist fences are useful when a thread is modifying persistent, yet local, state.

2.1.2 System Layout

Future scale-up servers are expected to have tens of sockets, hundreds of cores, and tens of memory controllers controlling terabytes of memory. In this work, we target a system that is divided into sockets, each of which has *cores* that generate stores, *memory controllers* or *MC*s that persist the stores, and a *gateway* node used for cross-socket communication (see Figure 2.1 and Figure 2.2). Note that successive stores from a single core may be persisted at different memory controllers; given a store's physical address there is exactly one controller that can persist it. The nodes in this system are connected by an asynchronous, loss-less, bounded-delay network with no out-of-thin-air values.

This distributed system is responsible not only for maintaining cache coherence but also for enforcing the persists-before ordering across distributed memory controllers. For correctness, at any instant in (asynchronous) time, those stores persisted across all memory controllers must form a consistent cut across the persists-before order. Vorpal algorithms represent this persists-before order using *vector clocks*.

2.1.3 Vector Clocks

Vector clocks [79, 28] are a well-known construct for representing logical time in a distributed system. A vector clock (or *VC*) is an n-tuple of positive integers, shared across n nodes. For a vector clock V, we say V_i denotes the ith entry in the vector. Vector clocks are useful in that they represent time as a partial order: every node's vector clock acts as a local clock periodically synchronized with remote clocks. With vector clocks, we can progress local time by *incrementing* the local vector entry by one. We express this operation as: $V +_i 1$, that is, an

increment adds 1 to ith slot of V. Orderings across node clocks are incurred upon synchronization. When the local node receives a message from a remote node, it updates its local clock to include all remote clock times prior to the message's send time, that is, it *merges* its local clock with that of the message's send time. We express the merge of vector clocks U and V as $U \oplus V = Z$ where $\forall_i Z_i = \max(V_i, U_i)$. As VCs describe a partial order, we can compare clocks easily. Given clocks U and V, $V < U \iff \forall_i V_i \leq U_i \wedge \exists_j V_j < U_j$; and $V = U \iff \forall_i V_i = U_i$. We can also define *simultaneity* as $V||U \iff \neg(V < U) \wedge \neg(U < V)$, when clocks U and V could occur in parallel.

2.2 Overview

Vorpal is a scalable, distributed mechanism for implementing persistence models. It uses vector clocks [79, 28] for encoding the persist-before orderings across stores and provides a scalable distributed algorithm for persisting stores at the memory controllers in accordance with those constraints.

The Vorpal algorithm class includes several variants, each of which is a refinement of previous versions. The simplest, Vorpal$_0$, is simple to implement but incurs extraneous orderings between persistent stores. Its refinement, Vorpal$_{chunk}$, operates at the level of a group of stores rather than single stores thereby supporting a more relaxed persistent order (necessary for higher performance). However, neither Vorpal$_0$ nor Vorpal$_{chunk}$ exploits the hierarchical nature of the network — they use very large vector clocks. To shrink the message size, we introduce Vorpal$_H$, a variant that maintains smaller clocks by leveraging hierarchy. However, a clock hierarchy has the tendency to introduce extraneous orderings due to sharing of vector clock dimensions between cores of the system. We reduce these extraneous orderings, while still leveraging hierarchy in Vorpal$_H$.

2.2.1 Definitions and Preliminaries

Vorpal algorithms comprise of two steps: *order construction* and *order enforcement*, each with their own requirements. We explain both, then use them to define correctness (both safety

and liveness) for the entire class of Vorpal algorithms.

2.2.1.1 Order Construction

In the order construction step, we assign vector clock values to persistent stores. The order built by these clock values, which we term the *clock order*, is expected to subsume the persists-before order, that is:

Definition 1 (Order Construction Requirement)**.** *A Vorpal algorithm satisfies the* order construction requirement *if given stores S and T with Vorpal-assigned clocks* V^S *and* V^T, $S \lessdot_{PB} T \rightarrow V^S < V^T$

In order to build the clock order, each CPU maintains a vector clock that will represent its local and monotonically increasing logical time (without loss of generality, we assume single-threaded cores). We use the term *core VC* to refer to the clock register maintained by each core to assign VC values to the persistent stores it generates.

2.2.1.2 Order Enforcement

In the order enforcement step, memory controllers persist stores according to their clock order (and, consequently, their persists-before order). The controller persists new store only if *all* preceding stores in clock-order have been persisted, that is:

Definition 2 (Order Enforcement Requirement)**.** *Consider the set of persisted stores Persisted and some store S with vector clock value* V^S*. A Vorpal memory controller that satisfies the* order enforcement requirement *will set* $Persisted' = Persisted \cup S$ *if and only if* $\forall_T V^T < V^S \rightarrow T \in Persisted$.

To do this enforcement, each memory controller maintains a vector clock to track the completion of stores. We use the term *progress VC* to refer to the VC register maintained at the memory controller. At any given time, the progress register *P* merges all persisted VC values it is aware of, including those persisted locally or at other MCs.

2.2.1.3 Correctness

Definition 3 (Safety)**.** *A Vorpal algorithm can be considered safe if at any given point in time, the set of persisted stores is a consistent cut across the persists-before order.*

Theorem 1 (Safety)**.** *Any Vorpal algorithm that meets both the Order Construction and Order Enforcement requirements is safe.*

Proof. By induction. **Base case**. $\mathit{Persisted} = \varnothing$. The empty set is a consistent cut across any partial order. **Induction**. We induct across store persists. Assume *Persisted* is a consistent cut across the persist-before order and some controller adds a store S such that $\mathit{Persisted}' = \mathit{Persisted} \cup S$. By order enforcement, S is only persisted $\forall_T V^T < V^S \rightarrow T \in \mathit{Persisted}$, and by order construction, $S \lessdot_{PB} T \rightarrow V^S < V^T$. Therefore, store S is added to *Persisted* only if $\forall_T T \lessdot S, T \in \mathit{Persisted}$. Since, for any partial order, the union of two consistent cuts (both *Persisted* and S and all its predecessors) is itself a consistent cut, $\mathit{Persisted}'$ is a consistent cut across the persists-before order. □

Definition 4 (Liveness)**.** *A Vorpal algorithm can be considered live if, given a long enough failure-free execution, it eventually persists all issued persistent stores.*

2.3 Vorpal$_0$: Baseline Vorpal

Having laid out the requirements for a Vorpal algorithm, we introduce Vorpal$_0$, a straightforward (though overly conservative) mechanism for meeting the two Vorpal requirements. Subsequent sections will refine the algorithm. Its code is shown in Figure 2.3.

2.3.1 Order Construction

In Vorpal$_0$, as with all Vorpal algorithms, we tag every store with a vector clock value generated by the core's source vector clock. In Vorpal$_0$, we build an over-constrained clock order. At every store, we increment the core's source VC (line 3). At every load-acquire, we merge

```
1  // Core_i State and Events (Order Construction)
2  clock_register C;
3  On: Issue persistent store or store−rel S
4    C +_i 1; V^S = C;
5  On: Issue store−rel S (non−persistent)
6    V^S = C;
7  On: Issue load−acq A that
8   synchronizes−with store−rel S
9    C = C ⊕ V^S;
10
11 // MC State and Events (Order Enforcement)
12 clock_register P;
13 On: Periodically
14   if(∃_S S ∈ Pending ∧ S ∉ Persisted
15         ∧are_adjacent(V^S, P)){
16     Persisted = Persisted ∪ S; P = P ⊕ V^S;
17   }
18 On: Received broadcast clock B
19   P = P ⊕ B;
20 On: Periodically
21   Broadcast P;
```

Figure 2.3. Vorpal$_0$: The pseudo-code for Vorpal$_0$, the baseline version of Vorpal. Actions of the cores and the memory controllers (MC) for order construction and enforcement, respectively, are shown.

the local VC with the VC value attached to the corresponding store-release (line 9). Trivially, Vorpal$_0$ meets the order construction requirement — it enforces program order on all stores (thereby capturing the persist fences of ARP) and preserves synchronization induced persistence orderings.

2.3.2 Order Enforcement

Vorpal$_0$'s memory controllers enforce the ordering created by the processors using a simple rule that leverages its local progress vector clock (P). Whenever a controller persists a store (line 16), or discovers the store has been persisted at another controller (line 19), it merges the store's VC to its clock. In order to ensure all stores are eventually persisted, controllers periodically broadcast their clocks to all other controllers (line 21), who merge the broadcasted values into their own local clocks.

To decide whether to persist a store, the memory controller uses the notion of *adjacency*. A memory controller persists a store S with clock value V^S only if the store's clock value is *adjacent* to the current progress clock P, that is, if $V^S_i \leq P_i$ for all i, except j where $V^S_j = P_j + 1$. Our notion of adjacency ensures that, for any persisting store, we only increment one dimension of the progress clock, and only by one.

We can show that Vorpal$_0$ meets the order enforcement requirement by induction on P,

where our hypothesis is that any store with a clock $V \leq P$ is persisted. When a controller persists some store S, any preceeding store $T \lessdot S$ must have a clock value smaller than V^S (by order construction), and, consequently, V^T must be less than or equal to P (by adjacency). Therefore T is persistent (by induction).

2.3.3 Liveness

Vorpal$_0$ is not only safe (by order construction and enforcement), but also live. We reason by contradiction: suppose some store S cannot be persisted, and has the smallest VC of any outstanding stores. Given a long enough failure-free execution, all persistent stores (including S) will be issued to their controllers. Given a long enough period with no persisted stores, all progress clocks will converge on a value P such that $\forall_{T \in Persisted} V^T \leq P$. By our contradiction assumption, S cannot be adjacent to P, which gives three impossible cases. If $S \leq P$, then we have violated safety. If $\exists_i S_i - P_i > 1$ or $\exists_{i,j} S_i - P_i = 1 \wedge S_j - P_j = 1$, then we violate our contradiction assumption as there exists some $S' \notin Persisted$ that persists-before S (due to the density of our order construction rule).

2.4 Vorpal$_{chunk}$: Chunk-Granularity Vector Clock Ordering

Vorpal$_0$ is correct, but introduces many extraneous orderings between stores. In particular, it assumes that all stores issued by a given thread must persist in program order, but this constraint is in fact not required by ARP. Rather, ARP assumes that a thread's stores are unordered into persistence unless constrained by a synchronization edge or persist fence.

In this next subsection, we introduce a relaxed version of Vorpal$_0$, called Vorpal$_{chunk}$, which groups together stores that are unordered with respect to one another and issued from the same thread. These groups of unordered stores, or *chunks*, are all assigned the same clock value. Like all Vorpal algorithms, the clock order captured by Vorpal$_{chunk}$ ($\leq_C$) subsumes the persist-before order, that is: $\lessdot \subseteq \leq_C$, and, furthermore, the controller in turn completes the chunks following the generated vector clock order (corresponding to $\leq_C$). We show its code in

Figure 2.4

2.4.1 Order Construction

In Vorpal$_{chunk}$, a chunk is created by incrementing the local clock register — all stores in the chunk will receive the same clock value. Once the chunk is completed, the dispatch function $DispatchChunk()$, sends the chunk, along with the count of stores inside the chunk, to its respective controller and resets the chunk counter used to track the number of stores (*ChunkCount*) inside a chunk.

A new chunk of stores is created on seeing new store following an earlier persist fence or load acquire (lines 29 and 28 respectively). In addition to these instruction triggered chunk boundaries, to simplify enforcement, Vorpal$_{chunk}$ creates a new chunk when created when the next store is issued to a different memory controller than the previous store (line 27). Cores also periodically start new chunks to maintain liveness (line 40).

Rather straightforwardly, Vorpal$_{chunk}$'s clock-order subsumes the persist-before order (and therefore meets order construction). Instructions that induce a persist-before ordering (either a persist fence or synchronization edge) result in the increment of the local clock register — the use of chunks is safe as no stores in a chunk are persist-before ordered with respect to another.

2.4.2 Order Enforcement

Vorpal$_{chunk}$ inherits its order enforcement code at the memory controller from the simpler Vorpal, but its code operates on chunks instead of stores. That is, if some pending chunk has yet to be persisted and is adjacent to the progress register P, then all component stores in the chunk can be persisted by the controller (line 44). Note the count of stores inside the chunk is made known to its controller by the core at chunk dispatch time, hence it can safely complete the individual stores and then mark the chunk as complete (so chunks can be sent piecemeal if necessary). As with Vorpal, the memory controller periodically broadcasts its clock to allow for cross controller progress (line 49).

```
// Core_i State and Events (Order Construction)
clock_register C;
Integer ChunkCount;
Set<Store> Chunk;
On: Issue persistent store or store−rel S
  if (S switches controller
   ∨S is first store after load acquire
   ∨S is first store after persist fence)
    C +_i 1; DispatchChunk();
  Add S to Chunk;
  ChunkCount++;
  V^S = C;
On: Issue store−rel S (non−persistent)
  V^S = C;
On: Issue load−acq A that
  synchronizes−with store−rel S
  C = C ⊕ V^S;
On: Periodically when ChunkCount≠0
  C +_i 1; DispatchChunk();

// MC State and Events (Order Enforcement)
clock_register P;
On: Periodically
  if(∃_Chunk Chunk ∈ Pending ∧ Chunk ∉ Persisted ∧
   are_adjacent(V^Chunk, P)){
    Persisted = Persisted ∪ Chunk; P = P ⊕ V^Chunk;
  }
On: Received broadcast clock B
  P = P ⊕ B;
On: Periodically
  Broadcast P;
```

Figure 2.4. Vorpal$_{chunk}$: The pseudo-code for Vorpal$_{chunk}$, a vector clock Vorpal algorithm that constructs and enforces ordering at a relaxed granularity of chunks. Actions of the cores and the memory controllers (MC) are shown.

For correctness of order enforcement, the proof previously laid out for Vorpal$_0$'s enforcement can be reapplied practically verbatim, substituting "chunks" for "stores". This substitution is possible because individual chunk is assigned a single clock value and stores that share a clock value are unordered. Similarly, the Vorpal$_0$ proof of liveness applies substituting in chunks for stores.

2.4.3 Vorpal$_{chunk+}$

Although Vorpal$_{chunk}$ has fewer extraneous orderings relative to Vorpal$_0$, the demarcation of chunks based on controller memory regions creates false orderings. Below we propose Vorpal$_{chunk+}$ which provides a way to remove these extraneous orderings through the use of *sub-chunks*.

In Vorpal$_{chunk+}$, we do not create chunk boundaries between persistent stores issued to different memory controllers but instead create *sub-chunks* when we switch controllers. Sub-chunks share a clock-value but are persisted at different memory controllers — a chunk is broken into a sequence of sub-chunks. With regard to order creation, this modification is safe, as stores

in different sub-chunks are still unordered with respect to each other in both clock-order and persists-before. However, enforcing the order becomes a problem as every component sub-chunk across all memory controllers needs to persist before any subsequent store is persisted.

To realize this optimization, each sub-chunk contains information about its successor and predecessor sub-chunks, along with a unique id. When the sub-chunk is persisted, it notifies its successor's memory controller but *does not* increment its local progress clock. Once the final sub-chunk both receives notification its predecessor sub-chunk has completed and its own sub-chunk has persisted, it increments its progress clock. The effect of this chained notification is that the final memory controller effectively persisted all the entire chunk. That Vorpal$_{chunk+}$ correctly achieves order enforcement is straight-forward: if two stores's are clock-ordered, the first's chunk will entirely complete across all memory controllers before any progress clock becomes adjacent to the second's clock. Similarly, the liveness arguments of Vorpal$_{chunk}$ directly apply.

2.5 Vorpal$_H$: Hierarchy-based vector clock ordering

The prior Vorpal variants capture and enforce the persist-before order with few false orderings, however, they are unlikely to scale to a many-socket machine. Since the algorithms require a clock entry per core, the vector clocks can get quite large and, more problematically, they ignore the organizational structure of the machine (see Figure 2.2). For programs that are NUMA-aware and avoid cross-socket communication, these earlier Vorpal algorithms can induce significant cross-socket overheads due to their flat designs.

In this section, we introduce the hierarchical variant of vorpal, Vorpal$_H$. Vorpal$_H$ uses a two-level clock hierarchy — a *global clock* across sockets and many socket-specific *local clocks*. The dimensions of the global clock grows as a function of the number of sockets, while the clock dimensions for the socket-specific local clock is a function of the number of cores inside a socket. Using hierarchy, Vorpal$_H$ achieves a reduction in the vector clock dimensions, a task necessary

to scale order construction and enforcement to large-scale systems.

2.5.1 Order Construction

In Vorpal$_H$, each store is labeled using a *hierarchical vector clock*. This more detailed clock consists of two vector clock times (the global and local clock), along with the store's socket of origin. We write the hierarchical clock U originating at socket n as the tuple (U_l, U_g, p). Hierarchical clocks generate a *hierarchical clock order*, written $\leq_H$. Given two clocks $U = (U_l, U_g, p)$ and $V = (V_l, V_g, q)$, the order ($\leq_H$) between the clocks can be stated as follows: $(U_l, U_g) \leq_H (V_l, V_g) \iff (U_g < V_g) \quad or \quad (U_l < V_l \quad and \quad U_g = V_g \quad and \quad p = q)$. Vorpal$_H$ uses the per-socket gateways to assign global and local clocks to stores. These gateways monitor traffic into and out of the socket, and are therefore a useful location for managing the clocks: the gateway contains a clock register (G) for the global clock and clock registers ($L[N]$) for all N cores in the socket.

The overall structure of Vorpal$_H$ is similar to the non-hierachical variants. Vorpal$_H$'s cores query their respective gateway for the global and local clocks. Upon receiving the clocks, the cores send the chunks they generate to the controllers along with the respective clock values. The controllers then complete the chunks respecting the $\leq_H$ order as specified by the hierarchical vector order. Figure 2.5 shows the Vorpal$_H$ algorithm. Like Vorpal$_{chunk}$, Vorpal$_H$ groups together unordered stores into chunks, each of which shares the same clock value. Like previous Vorpal algorithms, the boundaries between chunks are incurred at memory controller switches, persist fences, or load-acquire synchronization edges. Vorpal$_H$ further differentiates between *local chunks* whose stores are persisted by a memory controller in the local socket and *global chunks* which are persisted by remote memory controllers.

Vorpal$_H$ uses the hierarchical clocks to issue orderings based on the adjacent chunk type. When two successive chunks (in persists-before or program order) are local, it uses the local clock registers (lines 64, 78 and 81) to order them, thereby avoiding publishing the ordering beyond the socket. In contrast, for any two successive chunks in which one chunk is global, the

```
// Socket_p Gateway: State and Events
// (Order Construction)
clock_register L[N];
clock_register G;
typedef enum type = {LOCAL, GLOBAL}
type chunk_type = GLOBAL;
On: Received a first store event S after load−acq,
 persistent fence, or controller−switch
 from local core_i
  if(no remote stores or updates to G since
   this event fired last)
    L[i] +_i 1;
  else
    G +_p 1;
  type new_chunk_type =
   (destination_socket(S)==p)?LOCAL:GLOBAL;
  if(chunk_type==LOCAL ∧
       new_chunk_type==GLOBAL)
    Notify core to dispatch chunk and
     mark as last local chunk
  else
    Notify core to dispatch chunk
  chunk_type=new_chunk_type

On: Received a store−rel S from local core_i
  V_l^S = (L[i], G, p);
On: Received a load−acq A from local core_i
 that synchronizes−with local store−rel S
  L[i] = L[i] ⊕ V_l^S;
On: Received a load acq A from local core_i
 that synchronizes−with remote store−rel S
  G = G ⊕ V_g^S;
On: Received a clock request from local core_i
 to be assigned to a chunk
  Reply with clock (L[i], G, p);

// Core state and Events (Order Construction)
Integer ChunkCount;
Set<Store> Chunk;
On: Any load−acquire, store−rel, persistent fence
 or when a persistent store switches controller
 or on a first store after load acquire
  Forward the event to the gateway
On: Received gateway notification to
     dispatch chunk (bool mark)
  (V_l, V_g) = RequestClocksFromGateway();
  if(mark){mark_as_last_local(Chunk);}
  DispatchChunk();
On: Issue persistent store S
  Add S to Chunk;
  ChunkCount++;
On: Periodically
  Issue persist fence;

// MC State and Events (Order Enforcement)
clock_register P_g;
clock_register P_l;
On: Periodically complete chunks
  if(∃_Chunk Chunk ∈ Pending ∧ Chunk ∉ Persisted
   ∧are_adjacent(V_l^Chunk, P_l)
   ∧are_adjacent(V_g^Chunk, P_g) ){
    Persisted = Persisted ∪ Chunk;
    P_l = P_l ⊕ V_l^Chunk;
    if(is_last_local(Chunk)) {
      Persisted = Persisted ∪ Chunk;
      P_g = P_g ⊕ V_g^Chunk;
    }
  }
  if(∃_Chunk Chunk ∈ Pending ∧ Chunk ∉ Persisted
   ∧are_adjacent(V_g^Chunk, P_g)){
    Persisted = Persisted ∪ Chunk;
    P_g = P_g ⊕ V_g^Chunk;
  }
On: Received broadcast clock B_l
  P_l = P_l ⊕ B_l;
On: Received broadcast clock B_g
  P_g = P_g ⊕ B_g;
On: Periodically
  Broadcast both P_l and P_g;
```

Figure 2.5. Vorpal$_H$: The pseudo-code for Vorpal$_H$, a vector-clock hierarchy-based Vorpal algorithm designed for large multi-socket systems. Actions of the cores, gateways and the memory controllers are shown.

algorithm uses the global clock to order them (lines 64, 78 and 84).

Note the remote socket's gateway participates in Vorpal$_H$ on behalf of the cores of that socket — cores must query the gateway for their clocks. The constructed order as captured in the local and global clock registers is used to make the local and global clock assignments to the chunks at the cores (line 87 and line 100).

By reasoning about local and global chunks, we can demonstrate that Vorpal$_H$'s order construction is correct. Consider any two stores that are adjacent in persist-before. Their ordering is induced by either a persist fence or synchronization persist-before edge. Their corresponding chunks are also either local or global. If both are local, the ordering instruction (the persist fence or load acquire) will result in ordering by the local vector clocks in the gateway (line 64) before the second store. If one or both are global, the intervening ordering instruction will result in ordering by the global vector clock (line 66) before the second store. In either case, the stores are ordered by the clock order $\leq_H$.

2.5.2 Order Enforcement

Order enforcement in Vorpal$_H$ takes advantage of the fact that its clock-order is *nested*, that is, two or more local chunks may carry the same global clock but are ordered by their local clock.

For enforcement, Vorpal$_H$ maintains two progress registers at the controllers, one for the local clock (P_l) and another for global clocks (P_g). The controllers complete the global chunks in global clock order (line 122) and the local chunks nested within the global chunk also in their respective local clock order (line 113), updating the global and local progress registers respectively. When persisting local chunks, we delay updating the global chunk until all nested local chunks that share a global clock value are persisted. We accomplish this tracking using a single bit to *mark* the final local chunk before a global one (lines 72, 99, and 116).

Reasoning about global and local chunks also demonstrates that Vorpal$_H$'s order enforcement is correct. Trivially, global chunks (and their stores) are correctly persisted with respect

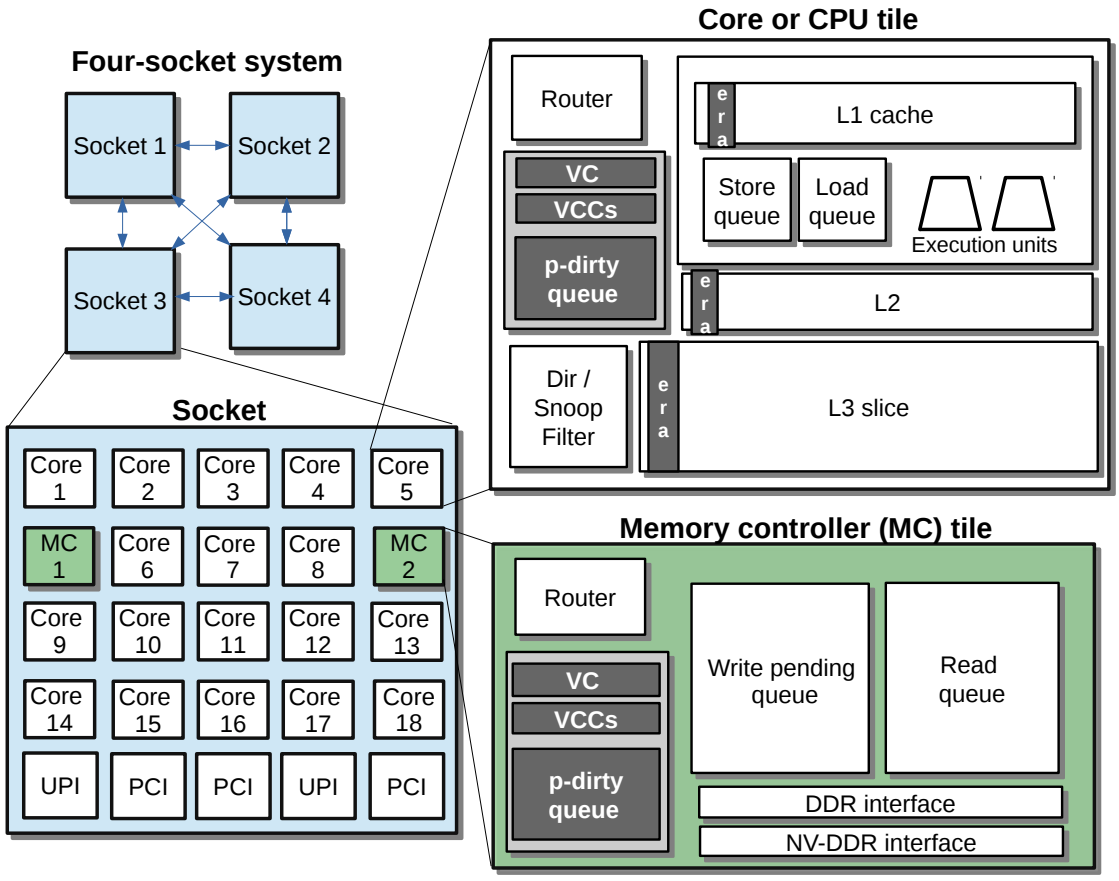


Figure 2.6. Detailed machine model: Representation on a tiled multi-core, muti-socket system.

to other global chunks. Similarly, local chunks (and their stores) are persisted in a consistent order with respect to other local chunks. Furthermore, local chunks are ordered with respect to subsequent global chunks through the marking mechanism, and are ordered with respect to earlier global chunks by the adjacency check on the global progress register.

Like all other Vorpal algorithms, Vorpal$_H$ is live. The proof of Vorpal$_{chunk}$'s liveness applies verbatim here, as we preserve the property that every clock increment corresponds to a distinct chunk.

2.6 Results

We first discuss the simulation infrastructure followed by the evaluation of Vorpal both with respect to performance and other benefits.

2.6.1 Methodology

We use MARSSx86 [86] simulator to model the effect of out of order (OOO) CPU. We use applications from Whisper suite for our evaluation [80], which are compiled and run on the simulator. With these workloads, the CPU generates persistent store traffic, and a set of memory controllers consume the generated store traffic. To model a non-volatile memory with slower write speed, we set the NVM write latency to 600 cycles (given the 2-4x slowdown expected relative to DRAM [25, 80]). We adapt the simulator to model a non-distributed ordering implementation wherein CPU coordinates the persistent ordering via synchronous messages with the memory controllers (similar to that described in [60]). We call this implemetation of ordering CPU-Sync. We also adapt the simulator to model our proposed Vorpal based distributed implementation for persistent ordering. We follow the ARP model for persistent semantics [61]. The sfences in the workloads are used to demarcate epoch boundaries. For the non-distributed implementation, controllers complete an epoch at a time and then send an acknowledgement(s). To evaluate the effect of controllers we vary the number of memory controllers as well as the number of sockets. We assume a 200 ns delay per inter-socket hop and topology with a single socket connected to at most two other sockets. We assume a 4 KB page size and spread consecutive physical pages across the memory controllers in a round robin fashion. We corroborated the thread interleaving in our model with the model in [87].

Figure 2.6 shows the representation of the modeled system. In the system model, persistent-dirty queues (p-dirty queues, in short) are used to store the pending dirty stores. Each p-dirty entry is also tagged with a VC. A CPU or a MC, besides maintaining a VC, also keeps a VC (called VCC) that represents the latest (as is knows best) VCs of the rest of the CPUs and

MCs in the system. The era bit inside each cache line is used to perform safe rollover of unique VC numbers. Using the bit, caches/directories can be queried to find out if any predecessor VC still exists in any of the p-dirty queue for a given persistent memory address.

2.6.2 Benefits of Vorpal

Figure 2.7 shows the results for CPU-Sync ordering solution versus Vorpal$_{chunk}$ by varying the number of controllers. The comparison is made relative to *ideal*. Ideal witnesses no persistence induced CPU stalls, i.e. the persistent stores are assumed to complete instantaneously. Vorpal's performance, as can be seen, approaches ideal with increasing controller count, by being able to exploit the parallelism offered by multiple memory controllers and deferred completion enabled by vector clocks. Most workloads benefit significantly from the proposed scheme (except for echo due to its very high store intensity). The results demonstrate the ability of Vorpal to provide significant performance and scalability benefits.

In Figure 2.8 we show the results of Vorpal$_H$ for a multi-socket setting with additional hop latencies. Large slowdowns are witnessed in a multi-socket setting for CPU-Sync ordering. Vorpal$_H$ due to its latency tolerance provides significant benefits in this setting. Compared to the slowdowns on a single socket system (Figure 2.7), the slowdowns witnessed are much higher. Many of the applications witness a slowdown more than 2X. CPU-Sync ordering is sensitive to the network topology, affecting some stores to remote memories more than others. We expect to see higher benefits from more relaxed models [61]. Vorpal$_H$ benefits occur in spite of the additional overheads of obtaining the global clocks which we account for in this experiment. Overall, the average speed-up obtained with all the improvements is 1.48X.

Figure 2.9 and Figure 2.10 shows supporting data on why delta and sparse encoded VCs will likely provide benefits. As can be seen in the figures, most of the time the delta values witnessed are less than 8 and for those above 8 most of lie fall under 16. A large fraction of the VC slots are just zeros. Influenced by this VC communication optimization opportunities, we adopt a sparse and delta encoding of VCs.

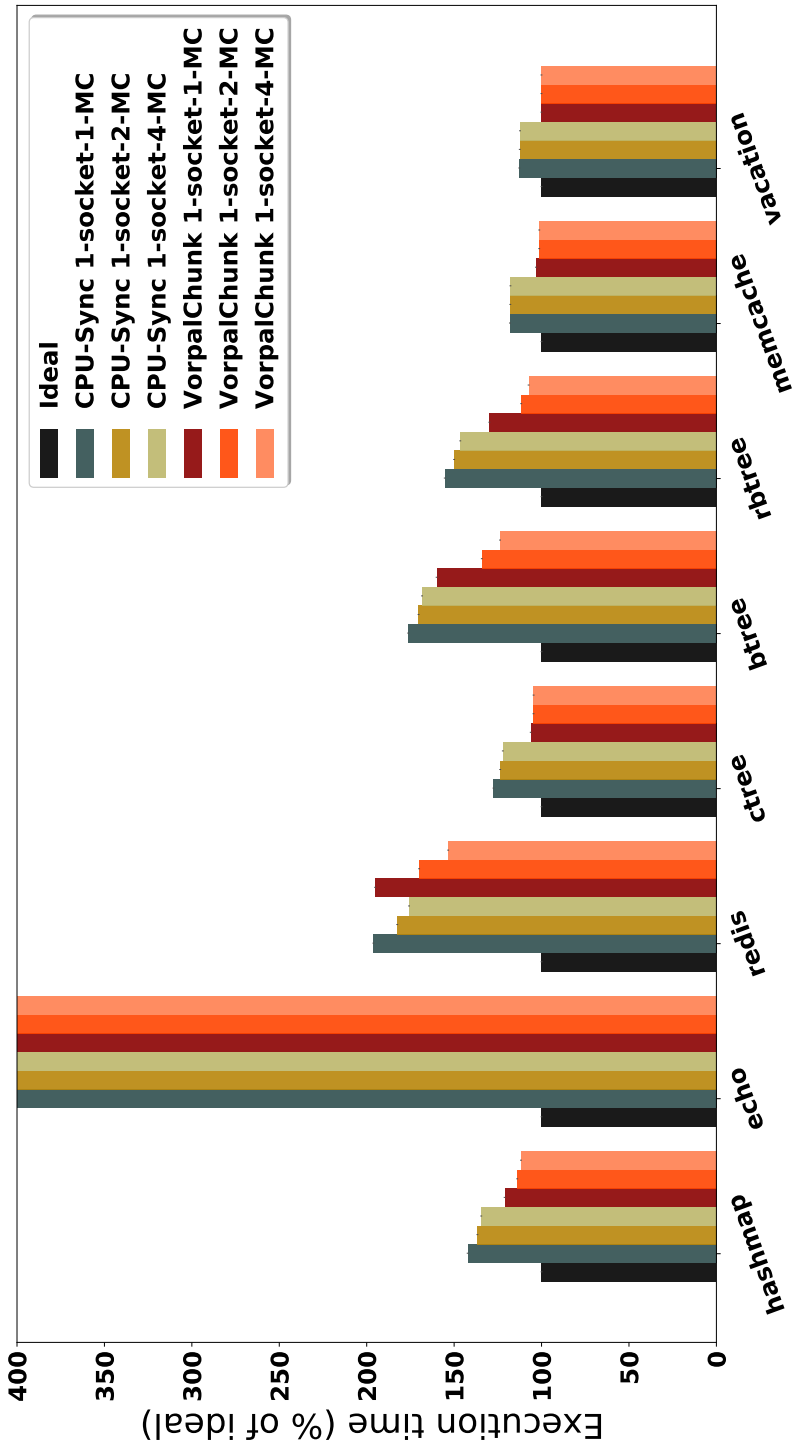


Figure 2.7. Vorpal$_{chunk}$ performance and scalability.

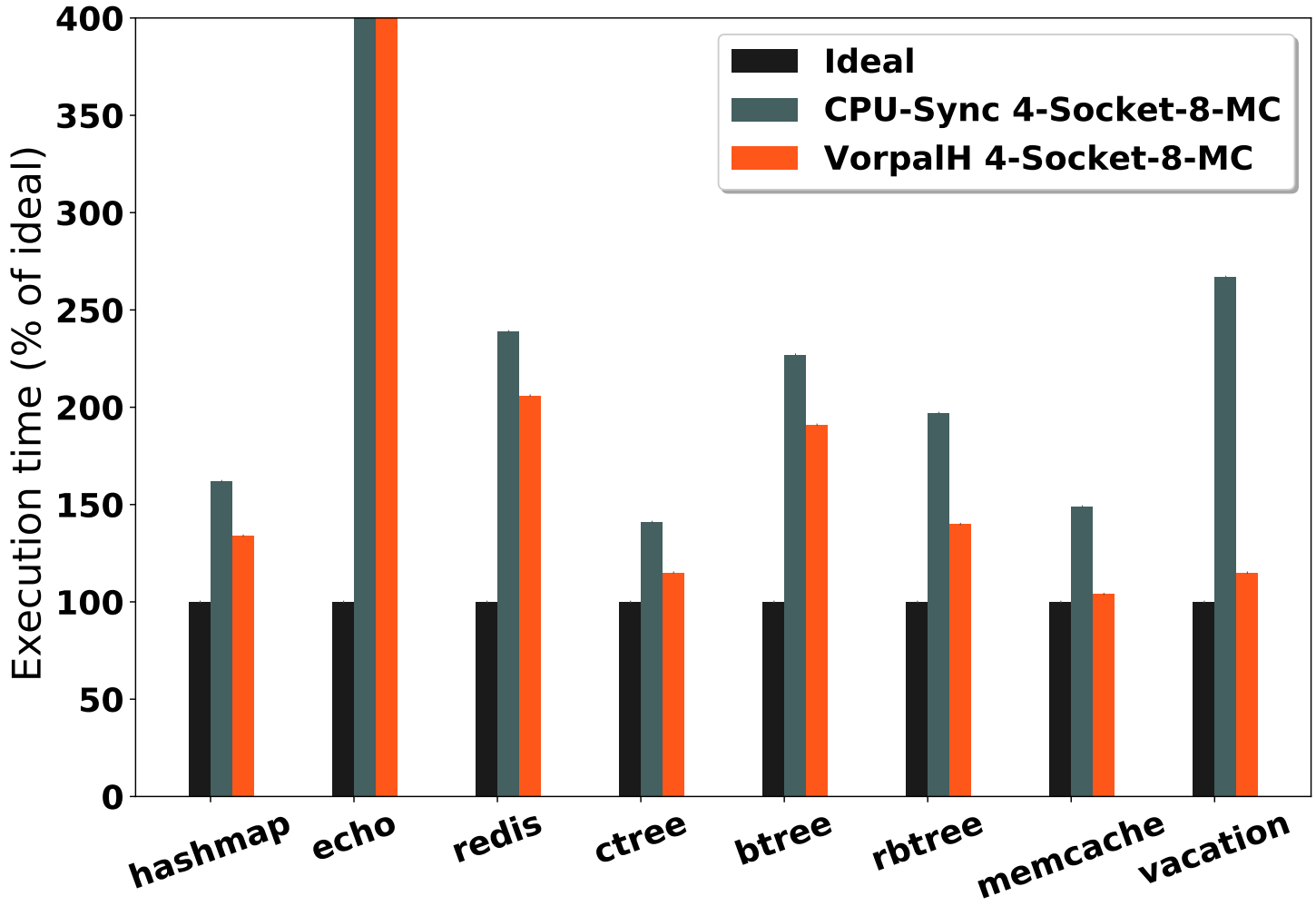


Figure 2.8. Vorpal$_H$ performance and scalability.

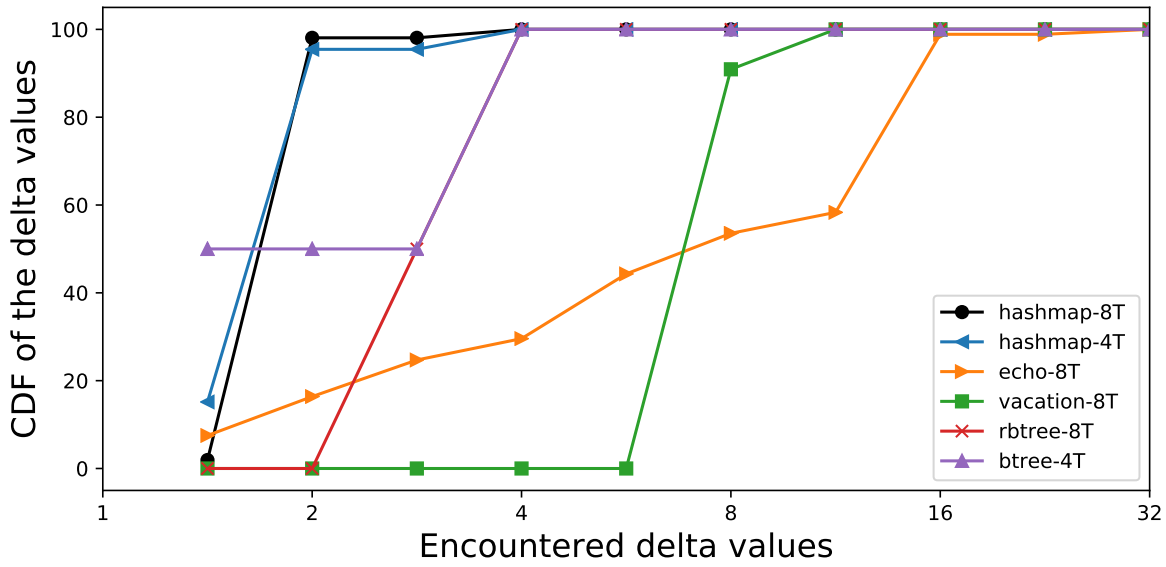


Figure 2.9. This figure shows the potential benefits of passing delta values of vector clock dimensions instead of absolute values.

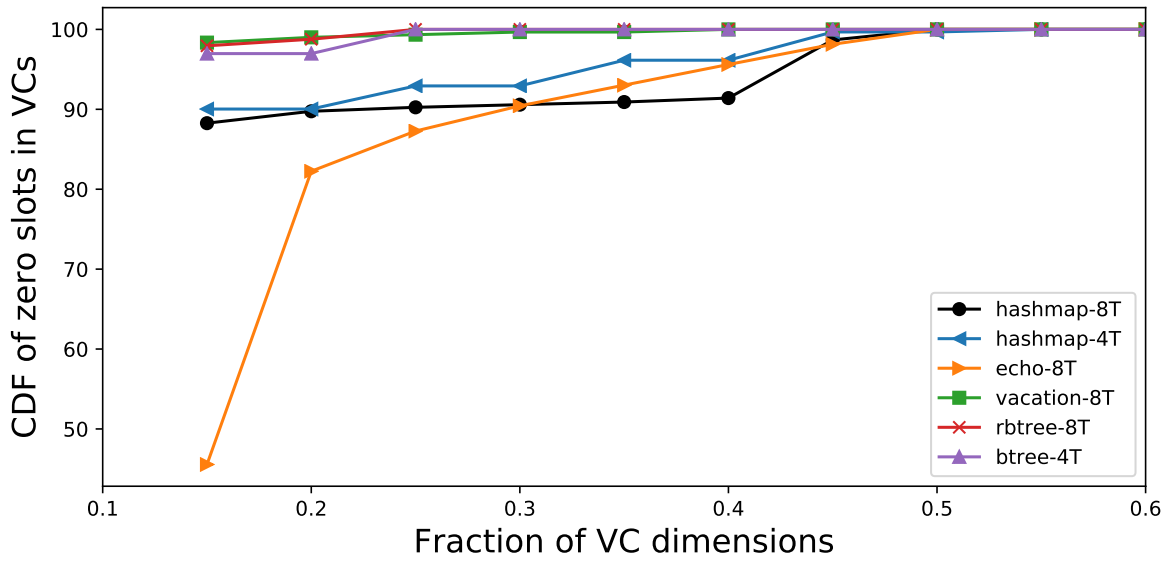


Figure 2.10. This figure shows the benefits of skipping certain dimensions of the vector clocks which have not changed in the interim.

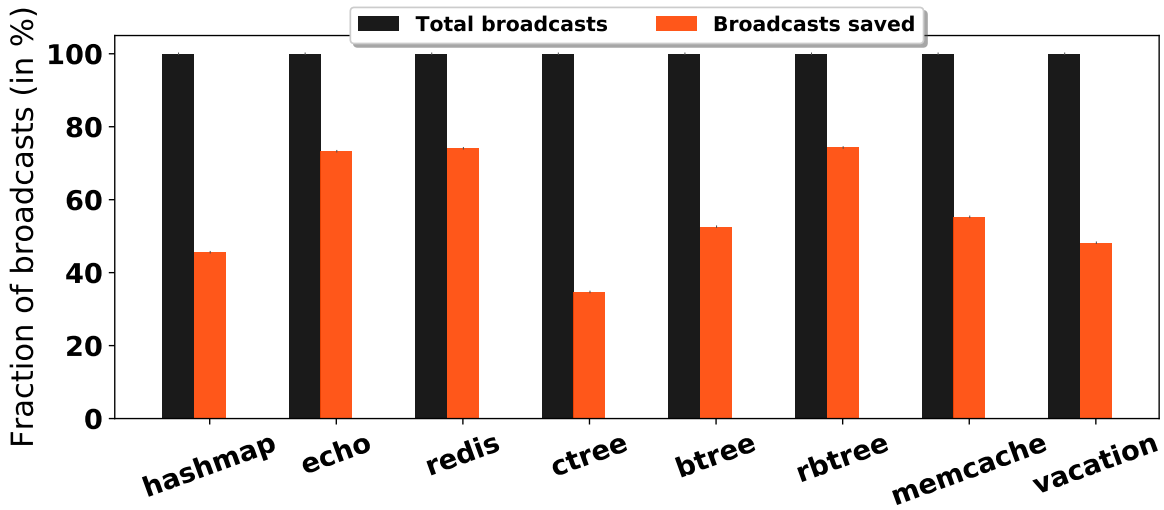


Figure 2.11. This figure shows the benefits in terms of broadcasts saved by Vorpal$_H$.

In Figure 2.11 we highlight the the extent of broadcasts saved by Vorpal$_H$. One of the benefits of the hierarchical ordering in Vorpal$_H$ is with regard to the broadcasts. Vorpal$_0$ induces large number of broadcasts and some of these are eliminated in case of Vorpal$_H$. As can be seen, each of the workloads benefit from this ability of Vorpal$_H$. The savings in the broadcasts is derived when the dependents of the store as per the VC order are destined inside the same socket.

2.7 Related Work

2.7.1 Memory Persistency

With the coming arrival of persistent memory devices, the research community has undertaken a significant effort in preparing various components of the system stack for their arrival. Vorpal sits alongside a number of architectural solutions for enforcing the persists-before order.

Most existing systems enforce persist orderings synchronous core-to-core or core-to-MC communication relying on unscalable and non-distributed algorithms [94, 5, 22, 87, 53, 60, 62, 61, 100, 80]. Neither scales to tens of MCs. Since ordering is enforced with some form of enqueue and acknowledgement messages between the CPU and memory controllers or CPUs,

these queue management operations introduce slow serializing communication. This design makes these solutions rather unworkable in modern multi-socket systems. Our proposed design specifically deals with this key challenge of scaling with multiple MCs through a scalable distributed algorithm.

Alongside the architectural community, the theory community has explored what persistent memory means at the language level, investigating novel persistency models [88] for specifying the persists-before order, e.g. [50, 89]. Work has also been done exploring both computational models [12] and model checking [45]. Other work has explored systems software level infrastructure, such as transactional memory-type abstractions [113, 20, 14, 49, 7, 73], file systems [121, 122], and data structures [82, 29, 6, 76, 112, 21].

2.7.2 Architecture Support

In this section, we describe existing architectural solutions to persistent write ordering in more details. We discuss their limitations in scaling persistence related communication and weaknesses in their ordering representations.

Existing ISA support from Intel provides explicit flush instructions [94] that forces cache lines to be synchronously sent to the memory controllers. By being stateless, this scheme entails low on-chip metadata. However, synchronous flushes makes this scheme perform very slow. It also requires software to name all affected cache lines before the fence hence complicating the programming model. Typically, memory models does not require explicit naming of cache lines.

Kolli et al. [60] proposed a mechanism to track persistent updates and perform enforcement without interfering with cache management and volatile execution. This means cache flushes need not be done synchronously (as prescribed by current ISA based solutions). Absent cache flushes, their design provides some performance benefits. Ordering constrains is derived based on the encountered programmer written fences (no cache line naming needed). The design however introduces non-significant on-chip metadata for each persistent cache line. A recent design [80], has proposed improvements to [60] which includes reduction in the metadata needs.

While the designs do address the issue of synchronous cache flushes [60, 80], they fail to contemplate the scalability issues in the presence of multiple memory controllers. Since ordering is enforced with some form of enqueue and acknowledgement messages between the CPU and memory controllers, these queue management operations introduces slow serializing communication between CPUs and MCs. This makes their solution rather unworkable in modern multi-socket systems. Our proposed design specifically deals with this key challenge of scaling with multiple MCs through a scalable distributed algorithm.

Condit et al. [22] have explored the possibility of using caches to buffer persistent updates and employing cache hierarchy for order enforcement. Their design associates each cache line with an epoch number and can support multiple inflight epochs. Here, an *epoch* is a group of unordered stores. By virtue of tightly coupling persistent order with cache lines and order enforcement with cache evictions, this design complicates cache replacement policies. Given the cache line and epoch association, there could be only one persistent cache line in flight at any given time for an address. Joshi et al. [53] proposed extensions to such a cache-based design, which includes inter-core dependency tables and eager eviction.

Fundamentally, while these designs manage to leverage cache based buffering of updates [22, 53], their ordering representation still have some key weaknesses. Firstly, their representation tend to create high on-chip metadata. Meta-data sizes being a function of the inflight epochs, total number of cache lines in the system and lifetime of updated dirty lines inside the caches. Further additional tables for inter-core dependency tracking increases costs, and cause stalls when tables are full. Our proposed ordering representation, on the other hand, is succinct in capturing both intra core program order and any synchronization induced inter-core dependency. This greatly simplifies implementation and makes the design amenable to further optimizations.

A recent work [100] has proposed hiding persist barriers related stalls through speculation. While using existing processor mechanisms is helpful, the extent of hiding is limited by the number of available per-core checkpoint structures.

2.7.3 Vector Clocks

Vector clocks were introduced independently in 1988 by Mattern [79] and Fidge [28] as a method of ordering events in a distributed setting. In this manner, the clocks improved on the original scalar clocks of Lamport [66], which could not represent the happens-before partial order of distributed events. Other methods of tracking logical time, including matrix time, have been proposed, for a summary of methods see [104]. The size of vector clocks has repeatedly been a concern, resulting in various methods of shrinking them (e.g. when FIFO channels are available [105]). Vector clocks have been used widely across computer science, showing up in transactional memory [93], databases [41], and checkpointing [120].

2.7.4 Time-based Memory Models

Lastly, although not directly related, it is worth mentioning about the use of logical time in memory system design. Recently, logical time has been shown to be useful in scaling coherence in both multi-core and GPU systems [72, 98, 103]. This paper, on the other hand, demonstrates a different use of time domain, specifically to support a scalable store-to-store ordering in the context of persistence.

Use of time dimension in memory system design has been explored in numerous ways [11, 31, 67, 92, 10, 116, 78, 77, 107, 1, 2, 72, 98, 103, 24]. The past work mostly focuses on scalable synchronization and/or coherence for large-scale multi-processors. They utilized time in different ways including interconnects that provide some guarantees on time [92, 11, 116]. Simplifying non-bus interconnects using the time dimension has been another line of work [1, 2, 78]: the main target was simplifying coherence on a range of interconnects. Given the distributed nature of constructing ordering and enforcing it we believe a distributed solution is warranted. The novelty in our work is on coming up with the right distribution of labor between the CPUs and MCs.

2.8 Conclusion

This paper brings the problem of ensuring the ordering of stores on large-scale persistent memory systems. Current solutions do not scale with increasing core and memory controller count. In order to fully exploit the benefits of emerging non-volatile memory technologies, we propose a distributed solution to the problem of persistent ordering inspired by vector clocks. Proposed algorithms provide scalability and performance (average speedup of 48%) while overcoming the limitations of applying vector clocks in this setting. We plan to explore even better algorithms and reflect on how the proposed implementations may impact persistent memory semantics as part of future work.

2.9 Acknowledgement

Chapter 2, in most parts, is a reprint of the material as it is accepted (conditionally) for publication in the Proceedings of the *38th Annual ACM Symposium on Principles of Distributed Computing (PODC '19)*; Vorpal: Vector Clock-Inspired Ordering For Large Persistent Memory Systems; Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao and Steven Swanson, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Scalibility in Caching

Scaling by combining caching policies with queuing theory

Despite advances in CPU architecture and DRAM memory technology, the memory wall continues to remain a bottleneck to application performance. The ever growing working set of applications exacerbates this problem even further. Increasing the capacity of the Last Level Cache (LLC) can help scale the memory wall. However, traditional SRAM based LLCs have limited density and high leakage power. Hence increasing the capacity of such LLCs is prohibitive in area, cost and power.

Emerging Non-volatile memories (NVM) based on technologies like Spintronics (e.g. Spin-Transfer-Torque (STT) RAM) and Resistive RAM offer energy efficiency improvement for normally-off computing [59, 32]. These memories also offer higher density and lower leakage power over SRAMs [119, 57], and hence are an attractive alternative to build large LLCs [115, 17, 3]. However, these technologies often suffer from long write latency that can degrade performance by causing write induced interference to subsequent critical reads. Additionally, the long latency of writes puts pressure on the request queues and the resultant congestion at the LLC can stall the core. As a result, LLCs based on NVM technologies underperform when compared to traditional SRAM LLCs for high performance applications.

Many techniques in devices, circuits and architecture have been proposed in both industry and academia to mitigate the performance impact of write latency as well as reduce write energy [115, 17, 3, 75, 58, 97, 83, 84, 126]. For instance, spin-hall-effect memory is another spintronic memory candidate that can reduce the write latency but at the cost of lower density [75, 58].

While technologists have been exploring material and device options to reduce the performance gap between emerging memories and SRAMs, circuit and architectural techniques have also been investigated. The write latency can be reduced by upsizing the write access transistors or using differential bits [97, 83] at the circuit level. However, these will result in higher overall power and lower density [57, 97, 19], thereby negating the density advantage offered by the emerging NVM technologies.

Another approach to reduce write latency is by relaxing NVM retention to enable low-energy and fast write but degrading reliability [55, 84, 106]. From industrial perspective, high performance on-chip NVM LLCs need to meet the stringent reliability requirement for high volume manufacturing (e.g. error rate less than 10^{-9} at temperature of 85C and above [18]). For example, to ensure correct write in some emerging memory circuit prototypes [84], write-verify operation is required which results in higher effective write latency. Lastly, some recent architectural techniques [115, 17, 3] propose intelligent ways to reduce the number of writes happening in NVM LLC. Our results and analysis show that even after applying these techniques there remains significant performance gap from an NVM LLC solution that can fully mitigate write latency impact while maintaining high density.

To summarize, multiple device and circuit techniques have been proposed to reduce NVM write latency, however these techniques either reduce LLC density or make NVM LLCs unreliable for industry adoption. Therefore, we need low cost architecture techniques to overcome the expensive writes without sacrificing the high density advantages the NVM technology offers. In this paper we propose new architecture for NVM LLCs that dynamically observes the write congestion at the LLC and mitigates the performance impact of long latency writes. Specifically we make the following contributions:

- We make the key observation that for NVM LLCs it is possible to trade-off some drop in LLC hit rate in order to aggressively bypass writes and still gain overall performance. To accomplish this, we propose write congestion aware bypass (WCAB) that uses a dynamic

learning mechanism in order to achieve an optimal bypass fraction in a given phase of execution, that would best balance the conflicting goals of relieving LLC congestion (through aggressive write bypass) and maintaining high LLC hit rates. We should note that WCAB is very different from traditional LLC bypass schemes where the motivation for bypass is to improve LLC hit rate by bypassing writes that have low priority of future reuse. WCAB, on the other hand, may end up reducing LLC hit rates while still delivering overall higher performance.

- We also observed that a lot redundant writes add to the LLC write traffic. We hence propose Virtual Hybrid Cache (VHC) that uses a portion of the L2 SRAM and the NVM LLC to preserve cache lines that frequently make trips between the L2 and the LLC. By intelligent placement and replacement management of cache lines, VHC eliminates a signifant fraction of writes.

- We perform detailed simulations and evaluate our policies on different LLC architectures (inclusive and exclusive) and evaluate the sensitivity of our proposals to varying NVM write latency and capacity. We show that, our policies gain significant performance at all configurations, while improving overall memory subsystem energy of the baseline system.

- Finally, we perform a thorough analysis of the tradeoff between the NVM write latency and density and show that a combination of circuit techniques to reduce write latency coupled with our architecture provides the best trade-off between density and latency for an NVM LLC design. By mitigating the long write latency, our proposals help the NVM LLC outperform a similar area SRAM LLC by nearly 18%, thereby enabling large NVM LLCs in the future that can deliver an SRAM like performance but at a significantly lower area cost.

3.1 Background

In this section, we first overview the characteristics of a representative NVM technology namely the Spin Torque Transfer RAM or STTRAM. We will then show that although STTRAM LLC can potentially provide significant capacity benefits owing to its high density, the interference caused by high latency writes makes it perform poorly compared to existing SRAM LLCs. This motivates the need for architectural techniques to mitigate the write latency in STTRAM LLC.

3.1.1 STTRAM based Last Level Cache

STTRAM offers density advantages over conventional SRAM as well as fast read and write time compared to other NVM technologies [119]. STTRAM utilizes a magnetic tunnel junction (MTJ) [57, 123], which is composed of a thin insulator layer sandwiched between a fixed ferromagnetic layer (polarization reference layer) and a free layer, to store the spin orientations as memory states (logic 0 or 1). The control of the MTJ states is through an access transistor, where the current flow through the MTJ can generate a spin torque based on the current direction and switch the magnetization direction in the free layer. The parallel and antiparallel magnetization states of the free layer (with respective to the reference layer) result in low resistance and high resistance states of the MTJ.

The main challenge for STTRAM based near logic memory application (e.g. LLCs) is its high write latency and high write energy compared with SRAM. This is due to the high thermal stability factor in STTRAM design, which is required not only for reliable operation but also for longer retention time to avoid refresh overhead. A typical thermal stability value of 60kT (k is the Boltzmann constant, T is the temperature) or up is desired [18]. In addition, for high yield requirement, a minimum design margin up to 6-sigma is required (failure probability of 10^{-9}) [18]. The write time of STTRAM bitcell increases with the thermal stability and 6-sigma design corner requires 2×-3× higher write time compared to low margin mean value corner [18].

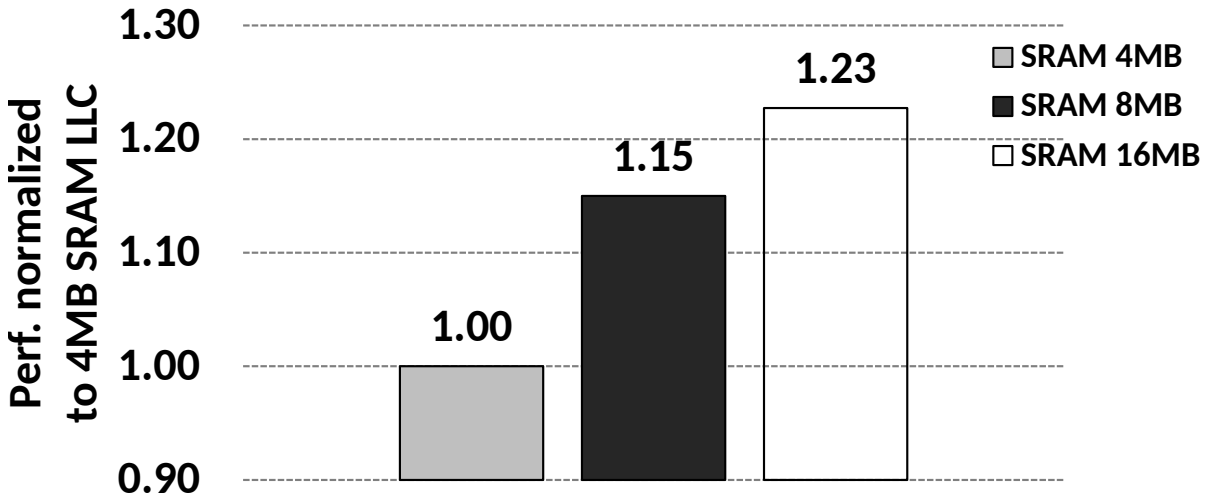


Figure 3.1. Performance benefit of higher capacity LLC

At 6-sigma design corner and 60kT thermal stability, a bitcell write time of more than 50ns is required [18]. Furthermore, retention time for a given thermal stability value degrades with temperature. At a realistic operation temperature of 85C or above, higher thermal stability (60kT or higher) is needed to maintain non-volatility. Therefore, long write latency of 10 to 50 ns is required for STTRAM to meet the high reliability LLC design requirement for industry adoption.

Although the write latency issue has been addressed in prior works, such as reducing the write pulse time [126], tailoring the access transistor sizes [75, 58] and relaxing the non-volatility with refreshing techniques [106], these techniques come at the cost of higher error rates and implementation overheads, and become even more challenging with advanced logic node scaling and increase in process variation [57, 97, 19, 18]. In this work, we focus on the architectural solutions which can meet industrial design requirements, and explore the design space of density and write latency trade-offs for STTRAM LLCs. We assume a write error rate (WER) less than 10^{-9} with SRAM-like ECC overhead and a thermal stability of 60kT.

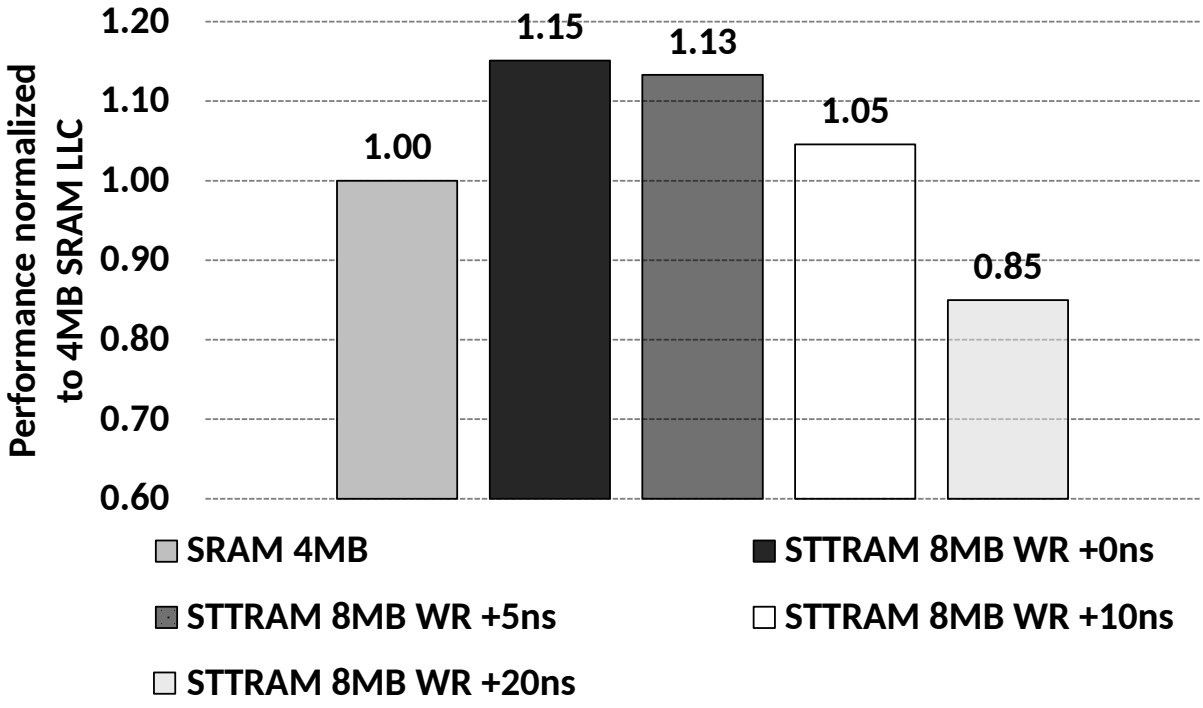


Figure 3.2. Impact of high STTRAM write latency

3.1.2 Write Latency versus Cache Capacity

Figure 3.1 shows the performance gains when the capacity of an SRAM LLC is increased from 4 MB to 16 MB, while the latency is kept constant, for a 4 core system 1 . On an average (geometric mean of all the 64 benchmarks described in the results section), 2× (8 MB) and 4× (16 MB) capacity increase brings 15% and 23% performance improvement over the 4MB baseline. These results clearly show significant performance potential of increasing the LLC capacity, which can be achieved by building the LLC using high density STTRAM technology.

Figure 3.2 shows the performance impact of increasing the write latency for an 8 MB STTRAM-LLC, compared to a 4 MB SRAM baseline. For this result we assume that STTRAM is 2× denser than SRAM and hence the 8 MB STTRAM LLC has the same area as the 4 MB SRAM LLC. We increase the write latency from 0 ns to 20 ns. As is evident from Figure 3.2, as the write latency of STTRAM increases, performance drops rapidly as long latency writes

interfere with reads. A hypothetical SRAM-like write latency STTRAM LLC (+0 ns) would gain 15% over the 4 MB SRAM baseline, which drops to 13% and 5% as the write latency is increased to 5 ns and 10 ns respectively. But when the write latency is further increased to 20 ns, all the capacity benefits are lost, and overall the performance drops by 15% when compared to the 4 MB SRAM baseline. As discussed earlier, lowering the write latency reduces the density advantages of STTRAM, and is not desirable.

In this paper, we hence explore architecture techniques to mitigate the long write latency and build an STTRAM LLC that can deliver SRAM like performance. We will discuss the existing solutions that have been proposed for STTRAM based LLC in the related works section. In the subsequent section, we will describe our proposed architecture in detail.

3.1.3 Sources of Writes in Last Level Cache

LLC architecture can be Exclusive, Inclusive, or Non-Inclusive [102]. Inclusive LLC duplicates every line in the inner level caches in the LLC. This helps simplify coherence flows, although it reduces capacity. However, as the L2 size continues to grow, LLCs are also being designed as exclusive [4, 34, 17], primarily because of the capacity benefits due to not replicating cache lines between the L2 and the LLC. Sometimes strict exclusion may not be possible for design simplicity, and such architectures are called as non-inclusive [17].

Different inclusion properties lead to different sources of write traffic at the LLC. Writes to an inclusive LLC originate from fills coming from the memory due to LLC read misses and dirty victims from L2. In exclusive LLC, both clean and dirty L2 victims are written to the LLC, while memory fills only happen to the L2. Write traffic in the exclusive LLC can be substantially higher as cache lines are deallocated from LLC on hits and are always written back when evicted from L2, irrespective of dirty or clean. On the other hand, in an inclusive LLC, clean L2 victims are dropped, as a copy of these cache lines is already present in the LLC. As a result of the higher write volume, designing an exclusive STTRAM LLC is more challenging, though exclusion brings in higher effective cache capacity. In this paper we will evaluate our proposal on both

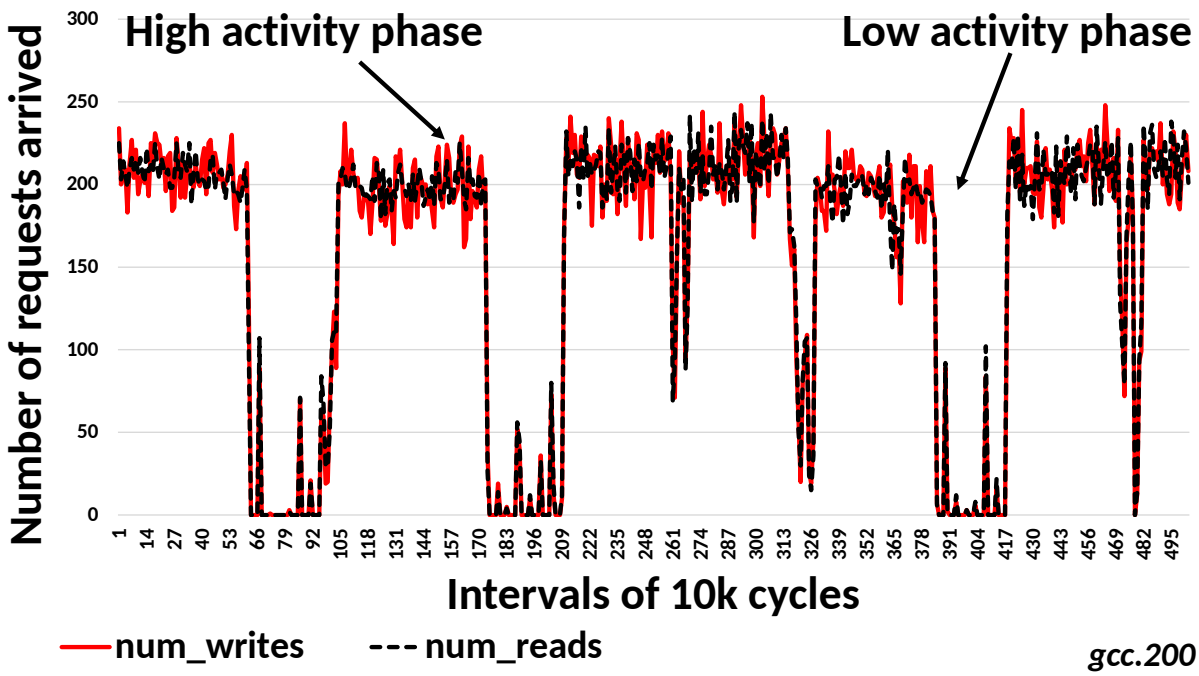


Figure 3.3. Request periods for a snippet of gcc.200

inclusive and exclusive LLC architectures.

3.2 Proposed Last Level Cache Architecture

Our proposal is composed of two main components - Write Congestion Aware Bypass and Virtual Hybrid Cache, which are explained in the subsequent sections.

3.2.1 Write Congestion Aware Bypass

One of the ways to reduce LLC congestion is to bypass some of the writes at the LLC. Many bypassing schemes have been proposed in the context of SRAM LLC [34, 56]. Traditional LLC bypass schemes employ a dead block predictor to classify lines which are less likely to be accessed again and therefore are not filled in the LLC. Bypassing such dead lines retains more live (more likely to be accessed again) cache lines in the LLC and therefore improves hit rate.

In the case of NVM LLC, bypassing not only improves hit rate but also reduces write congestion, thereby having a greater impact on performance. Bypass policies proposed in [115,

3] adapt SRAM bypass schemes to NVM LLC and demonstrate superior performance. Unfortunately since these bypass schemes are inherently designed for improving hit rates, the amount of bypass accomplished by them is fairly limited. Moreover, as the LLC capacity grows, the fraction of writes that will not be reused drops as larger capacity, enabled by high NVM density, allows more cache lines with large reuse distances to be retained. As a result, more aggressive bypass policies are needed to relieve the LLC congestion because of long latency writes. Unfortunately just increasing the aggressiveness of bypass can significantly affect LLC hit rates, thereby negating the capacity benefits offered by the NVM LLC.

We hence need to strike a balance between the conflicting goals of bypassing writes to relieve LLC congestion and the need to minimize the hit rate loss in the LLC because of bypass. To understand this trade-off better we first try to analyze our workloads and develop an understanding of the parameters that effect this trade-off. Based on this learning, we will then propose a novel algorithm that we call as the Write Congestion Aware Bypass (WCAB) (described later).

3.2.1.1 Write Congestion versus Hit Rate

For NVM LLC, the bandwidth delivered by the LLC depends on the write latency as long write latency blocks an LLC bank from accepting future requests, while the write is being performed to the bank. Reducing the latency of the writes or the fraction of writes will improve the NVM LLC bandwidth, thereby reducing queuing latency at the LLC and improving performance. As write latency is fixed for a given configuration, we need to reduce the fraction of writes by write bypassing. Unfortunately write bypass will result in increased misses to memory, thereby impacting performance. The cost of bypass is dictated by the latency of the main memory and the fraction of write bypasses that will result in future memory reads (liveness of the application). Slow memories can increase the cost of a wrong bypass. Likewise application phases with high liveness will suffer more from wrong bypass.

Figure 3.4 shows the performance (left Y axis) and hit rate loss (right Y axis) for

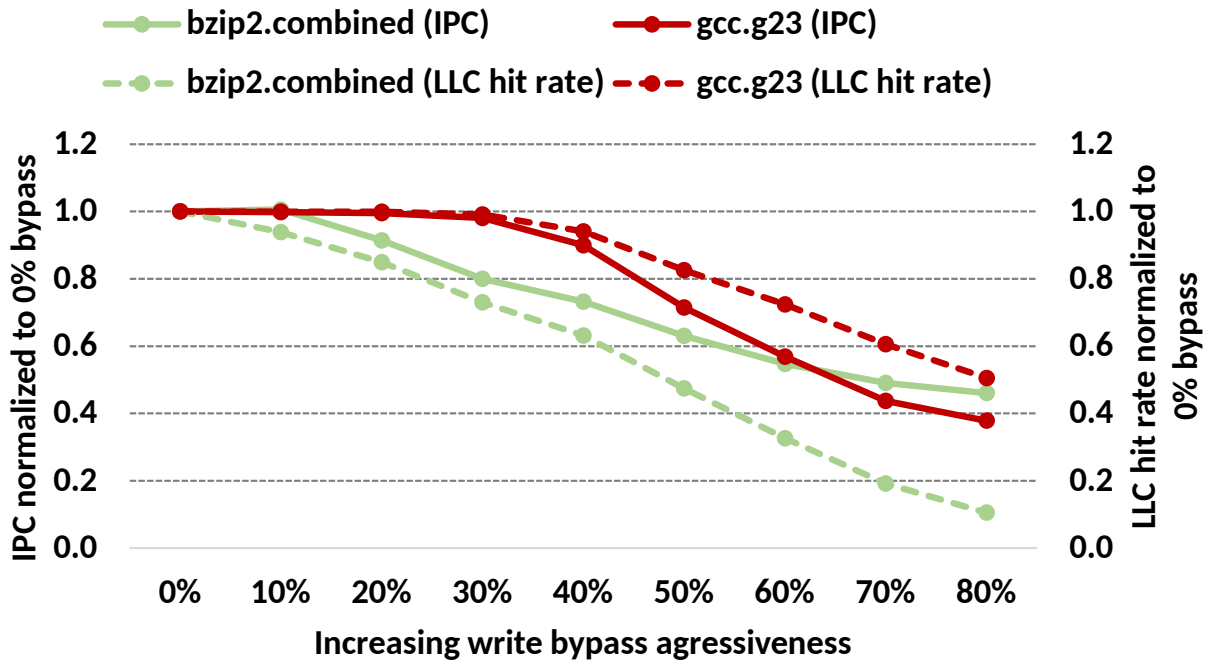


Figure 3.4. Impact of write bypass on SRAM LLC

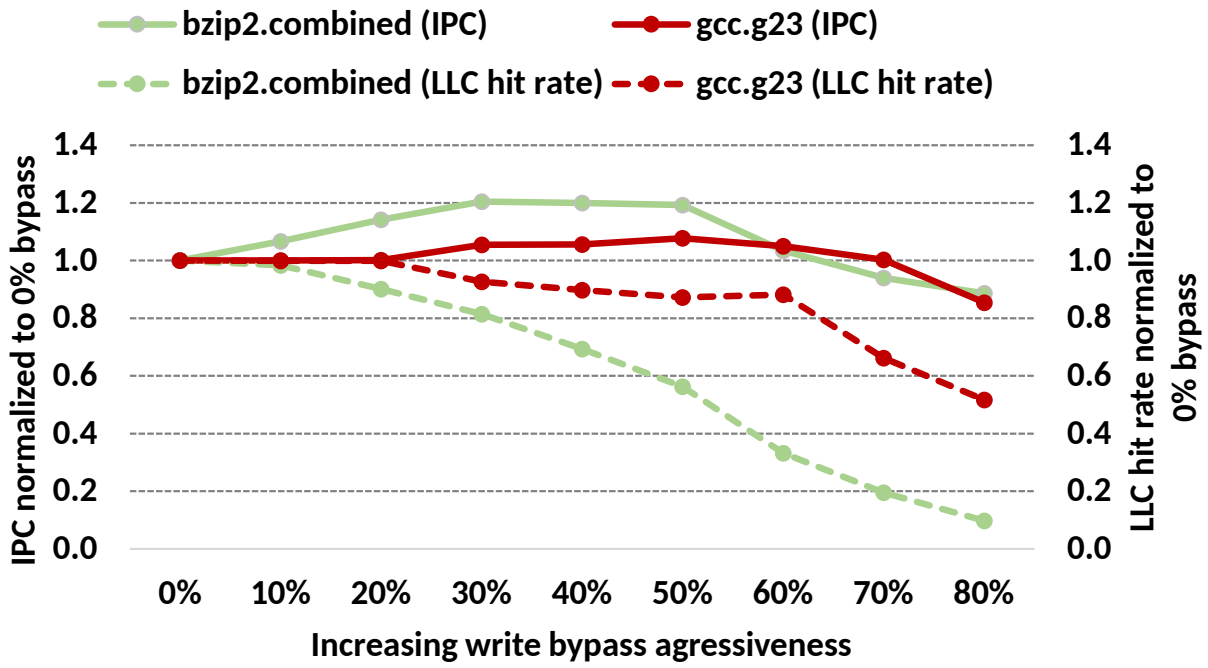


Figure 3.5. Impact of write bypass on STTRAM LLC

two representative benchmarks bzip2.combined and gcc.g23, as the bypass aggressiveness is increased for an SRAM LLC. Figure 3.5 shows the same graph when applied to an STTRAM LLC. In case of SRAM we clearly see that as the bypass aggressiveness is increased, the hit rate drops and overall performance degrades. This is expected because SRAM does not have any congestion and hence bypassing is detrimental to hit rate and consequently to performance. For STTRAM in bzip.combined we see that, like in the case of SRAM, the hit rate continues to drop as the bypass fraction is increased. Interestingly, despite of the hit rate loss, the performance initially increases as the bypass helps relieve LLC congestion. However beyond a point, the loss in hit rate outweighs the improvement in queuing latency and the performance eventually drops. Similar behavior is seen for the gcc.g23 workload. This clearly matches our intuition that a we need to derive an optimal bypass that balances the conflicting goal of capacity management and reducing LLC congestion.

To summarize, our goal is to find an optimal bypass that reduces the LLC queuing latency while minimizing the cost of bypass. Combining all the learning above, we can recapitulate that the optimal bypass depends on request bandwidth demand, fraction of writes, write latency of STTRAM, main memory latency and the liveness of the application. Of these, the latencies at the LLC and the memory are fixed for a given system design, whereas the liveness, write fraction and request bandwidth need to be learnt dynamically, for a given phase of execution of an application. We now describe our proposal, write congestion aware bypass (WCAB), that learns these parameters through a simple lightweight learning mechanism, and then uses it to modulate the fraction of bypass.

3.2.1.2 Write Congestion Aware Bypass Algorithm

Our LLC controller models a Request Queue, which stores LLC requests as they arrive. Requests are taken out of the request queue in order and sent to the LLC. Misses and victims may need subsequent passes through the request queue.

(A) Learning LLC congestion.

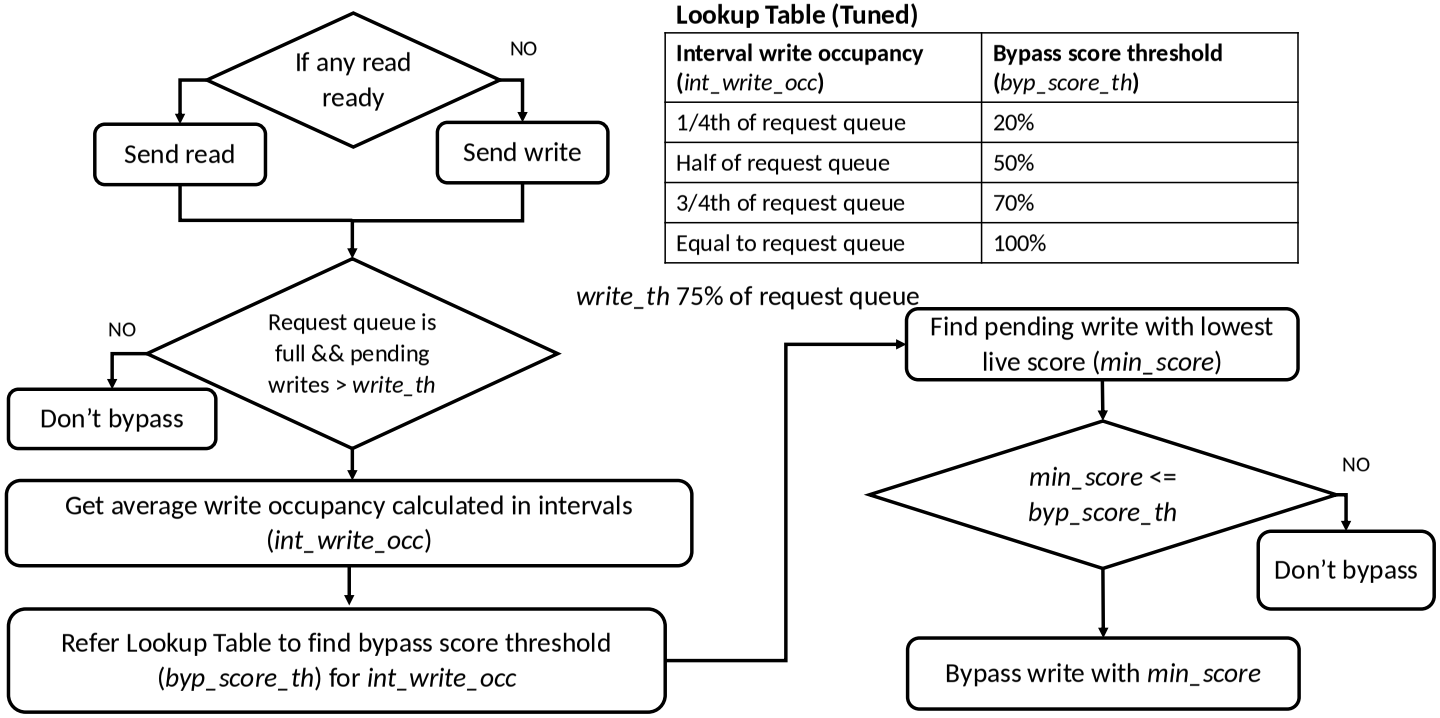


Figure 3.6. Detailed flowchart of the WCAB algorithm

If the request bandwidth de- mand is higher than the bandwidth supplied, the request queues will have a high occupancy. Hence the occupancy of the request queue can be used as an indication for LLC congestion. We should note that the request queues can also have high occupancy because of a large number of pending memory requests. To differentiate from this scenario, we also look at the number of writes pending in the request queue. If the occupancy of the request queue is high and the fraction of writes is above a threshold (write th), we infer it to be a scenario for LLC congestion. We keep a running counter that counts the average number of writes as well as the average occupancy seen in previous K windows (int write occ). Based on the values of these counters we decide the amount of bypass that needs to be done in the current window. Windows of high write congestion will have more bypasses, and those of low congestion will have fewer WCAB induced bypasses. A running average also makes sure that small spikes in bandwidth demand are not treated as phases of LLC congestion and unnecessary bypass is prevented. We found K = 5 and interval window length of 100K cycles, to be the best suited for our work.

(B) Learning the Cost of Bypass.

Write bypass will reduce hit rates and increase traffic at the memory. To estimate the cost of bypass, WCAB learns the reuse probability (liveness) of a given write that needs to be allocated at the LLC. We allocate small number (32) of observer sets in the L2 and the LLC, similar to [34, 56] for learning. A table indexed by hashed L2 access PCs (Instruction address that last accessed the cache line in the L2) is maintained at the L2. The table has four 10b signed, saturating counters corresponding to the liveness buckets 0-20%, 21-50%, 51-70% and 71-100% (we experimented with more buckets, but did not see much sensitivity). We define liveness as the fraction of writes corresponding to an access PC that are recalled from the LLC. So for instance, 20% liveness would mean that out of every 100 evictions from an access PC in the observer sets that were filled to the LLC, 20 were hit by subsequent reads. The observer sets are chosen similar to [34, 56].

Whenever a cacheline is evicted from the L2 observer set we decrement the liveness counters corresponding to its access PC. Note similar to the sampler of [56], we maintain partial-PC tags for lines in observer sets. If the line is recalled from the LLC in the future (LLC hit), we increment the liveness counters of the access PC. For example, for the 21-50% liveness counter, on eviction we decrement by 2 and on a recall we increment the counter by 10 (corresponding to 20%). If this counter is positive, it will mean that the liveness was at least 20%.

Whenever a cacheline is filled in the non-observer sets of the L2, we check the live counters corresponding to the eviction and assign it a 2 bit live score. This live score is stored for each L2 cacheline. The live score is basically the highest liveness bucket that was positive for its access PC. For instance if both 71-100% and 51-70% liveness counters are positive, it is assigned the 71-100% live score. Since we are only tracking four buckets, we need just 2 bits. On eviction, the live score is sent along with the write to the LLC and is stored in the LLC request queue. We should note that the default live score for all writes (when none of the counters are positive) corresponds to the 0-20% liveness bucket.

(C) Write Bypass.

WCAB checks the LLC congestion counters and based on the congestion decides a target live score (*byp_score_th*). Writes that have a live score below the target live score are bypassed. The bypasses are done as long as the occupancy of the request queues do not drop below a threshold, that is, bypasses are done as long as there is write congestion. If the amount of congestion is high, WCAB has a higher target live score. If the congestion is low, we do conservative bypassing by reducing the target live score. WCAB always ranks pending writes in order of their live scores. Writes with low live scores are bypassed before writes with high live scores. This helps retain writes that had higher likelihood of producing hits in the future, thereby minimizing the hit rate loss because of bypassing potentially live writes. Figure 3.6 depicts the WCAB algorithm in more detail.

(D) Tuning the Thresholds.

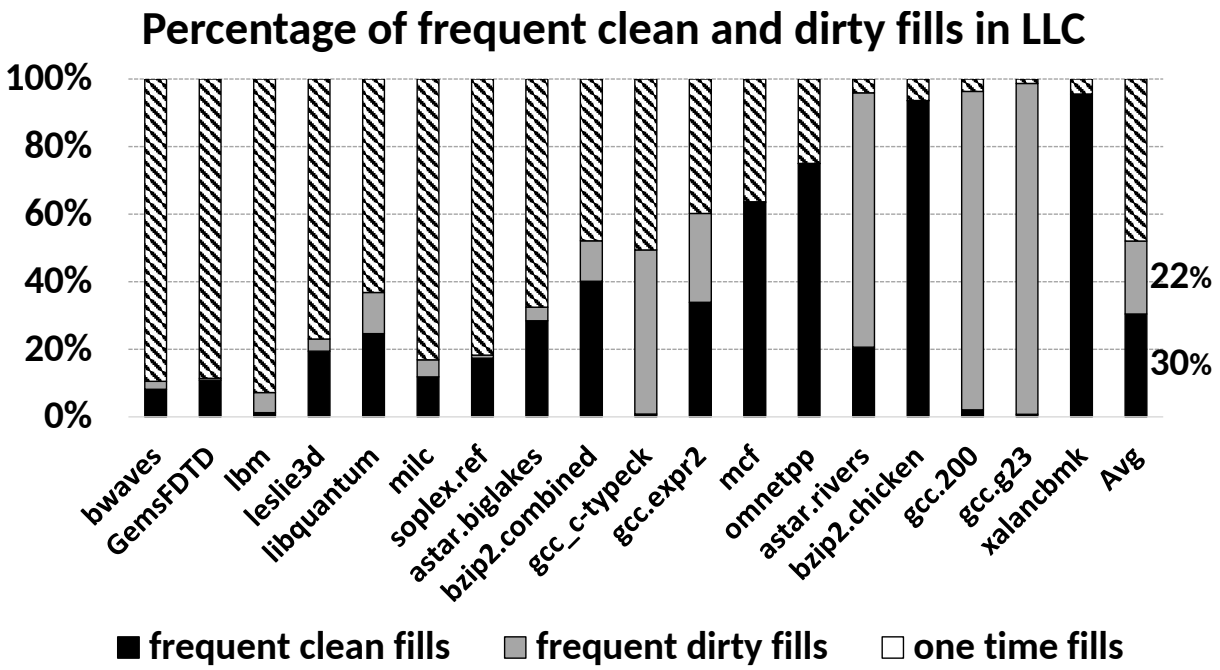


Figure 3.7. Frequent clean and dirty fills in LLC

The amount of bypass and hence the corresponding thresholds depend on the write latency of the LLC and the latency of the main memory. Therefore the thresholds need to be tuned for a given system. We ran various simulations for our target system to identify the best average thresholds. Figure 3.6 also shows (in the table on top right) the tuned parameters for our target system.

3.2.2 Virtual Hybrid Cache

In inclusive LLCs, many cache lines are repeatedly recalled from the LLC, modified in the L2, and written back to the LLC. We call such cache lines that frequently move between the LLC and the L2 as frequent dirty fills. In exclusive LLC, a read hit deallocates the cache line and moves it to the L2. On an L2 eviction, this line has to be written back to the LLC, irrespective of whether it was clean or dirty. A subsequent hit will move this line back to the L2. Therefore exclusive caches also tend to have significant amount of frequent clean fills. Figure 3.7 shows the average (arithmetic mean) percentage of such clean and dirty frequent fills for an exclusive

LLC. We see that such fills contribute to a significant portion of the overall writes and can be as high as 50% of the total writes at the LLC.

To reduce frequent fills at the NVM LLC, we propose a solution that we call as the Virtual Hybrid Cache (VHC). Unlike true hybrid caches proposed in [115] that use a dedicated SRAM cache for such frequent fills, the VHC simply borrows some capacity from the L2 and the LLC. To reduce writebacks because of dirty L2 evictions, VHC retains cachelines, that create frequent dirty fills, in the L2 so that multiple L1 writebacks may merge in the L2. For exclusive caches, VHC duplicates some lines in the LLC for reducing frequent clean fills. A recent proposal called LAP [17], attempts to tackle the clean eviction problem with exclusive caches using a similar approach. However, unlike LAP, VHC minimizes the LLC capacity loss because of duplication, through smart optimizations in the LLC. We next describe these two components of VHC in more detail.

3.2.2.1 Write Merges in L2

To reduce frequent dirty trips in the LLC, the cache lines classified as getting frequently dirty are given preference to stay in the L2. To classify frequent dirty lines we used the predictor proposed in [115]. By keeping the frequent dirty lines in L2 for longer, many writes merge in the L2 without the need of making fills in the LLC. To implement this technique, we keep a count of how many frequent dirty lines are in each L2 set. If this count is lower than W ways, then the frequent dirty lines do not participate in victim selection. Otherwise, if the count is equal or higher than W, then all the lines, including frequent dirty lines, participate in victim selection. For our workloads and configurations, W of 2 performs the best. This means we end up dedicating up to 2 ways for frequent dirty lines. Note that this trades off some L2 hit rate as some ways are dedicated to frequent dirty lines and therefore clean lines are evicted to the LLC more often.

3.2.2.2 Relaxed Exclusivity

As described earlier in the background section, in exclusive LLCs even the clean victims from L2 are filled in the LLC, creating write congestion problem due to frequent clean fills in NVM LLC. One solution, as proposed in LAP [17], is to not deallocate lines from the LLC on a read hit thereby duplicating some lines between the L2 and the LLC. This will eliminate all the clean fills after their first trip. However, the disadvantage of such a scheme is the LLC capacity loss due to duplication. To minimize the hit rate loss from duplication, we propose the following optimizations.

- We retain the duplicated lines in the LLC closer to the LRU position so that the replacement policy in the LLC does not degrade. If a line moves from the LLC to L2 and is retained in the L2 for a long time, it will reduce effective capacity because of duplication for a long time. To prevent such lines from being duplicated in the LLC, we do not update the LRU on duplication. The LAP policy of [17] on the other hand, moves duplicated lines to the most recently used (MRU) position to reduce clean fills. As a result our policy reduces clean fills by a lesser amount as compared to LAP, but has a lower hit rate impact on the LLC.

- If a cache line is hit by a store request in the LLC, then there is no value in duplicating the line as it would be made dirty in the L2 and the LLC copy will be stale. Hence such lines are not duplicated in the LLC. However, there are many cases when a read request brings the line into the L2, and a subsequent store makes this line dirty in the L1. In such cases, as soon as the line get dirty in L1, we send a hint with the coherence packet to de-allocate its copy in the LLC, thereby reducing an unnecessary duplication.

Coherence in VHC is handled similar to the exclusive LLC. Exclusive LLCs typically keep a snoop filter, which stores owner core for all the cachelines filled in core caches. For a duplicated line between the L2 and the LLC, the corresponding L2 is stored as owner in the

snoop filter. The LLC copy is not regarded as latest.

3.2.3 Area Overheads

For WCAB, 2 bit live score is stored for each L2 cacheline and assigned for every write to LLC. For 256KB L2 and 64 entry LLC queue, additional storage of around 1 KB per core is required. We use 256 entries in the hashed table (10 bits per PC) and 4 10 b counters. That needs an area of 1.5 KB. Duplication of lines in the L2 or L3 would need a bit to store the duplication in the L2. This needs 0.5 KB of L2 area. For observer sets (32), partial-PC tags account for another 0.5KB area. Overall our architecture adds an additional 3.5 KB of area per core. As compared to the 8 MB LLC and 256 KB L2, this constitutes less than 0.1% of the total cache area.

3.3 Results

We first discuss the evaluation methodology followed by the results.

3.3.1 Evaluation Methodology

For our simulations, we model four dynamically scheduled x86 cores with an in-house modified version of the Multi2Sim simulator [111]. Each core is four-wide with 224 ROB entries and clocked at 4 GHz. The core microarchitecture parameters are taken from the Intel Skylake processor [23]. Each core has 32 KB, 8-way L1 instruction and data caches with latency of three cycles and a private 256 KB 8-way L2 cache with a round-trip latency of eleven cycles. In our SRAM baseline configuration, all the cores share a 4 MB, 4 banks, 16-way LLC with round-trip latency of twenty cycles. For STTRAM configurations, we increase LLC capacity to 8 MB (with 8 banks) and add an additional 20ns of write latency. We shall also sweep this write latency and study the sensitivity of write latency to our policies.

Each core is equipped with an aggressive multi-stream stride prefetcher that prefetches into the L2 and LLC caches. The main memory DRAM model includes two DDR4-2400

Table 3.1. Benchmark selection and characterization.

Benchmarks	LLC MPKI
gcc.g23, xalancbmk, gcc.200, astar.rivers2, bzip2.chicken, gcc.c-typeck,omnetpp	0-10
libquantum, gcc.expr, leslie3d, bwaves, GemsFDTD, milc, mcf	10-20
soplex.ref, bzip2.combined, astar.rivers8, astar.biglakes, lbm, hpcg	20-100

channels (total bandwidth 38.4 GB/s), two ranks per channel, eight banks per rank, and burst length of eight. Each DRAM device has a 2 KB row buffer with 15-15-15-39 (tCAS- tRCD-tRP-tRAS) timing parameters. An additional ten-cycle I/O delay (at 1.2 GHz) is charged for each access to account for floorplan, board delays, etc. Writes are scheduled in batches to reduce channel turn-arounds.

We select 20 workloads from the SPEC CPU 2006 [8] and HPCG [43] benchmark suites. These are traces that have high L2 MPKI (more than 10) and will hence be sensitive to the LLC optimizations proposed in this paper. Table 3.1 lists these workloads along with their LLC MPKI. We first created 20 homogenous, RATE-4 traces (each core runs the same copy of the benchmark) for these workloads. In addition to these 20 homogeneous multi-programmed mixes, we prepare 44 four-way heterogeneous, multi-programmed mixes by randomly combining these 20 traces. In all these 64 multi-programmed mixes, each core simulates 250 million dynamic instruction and all the cores combined together simulate at least one billion instructions. Cores that finish early continue to run. We use weighted speedup [52] as the metric of performance.

3.3.2 Simulation Results

In this section, we evaluate the performance of our proposals. We first describe our baseline LLC policy and then summarize the performance gain of our features on top of this baseline for both exclusive and inclusive STTRAM LLC. After that, we will present a detailed

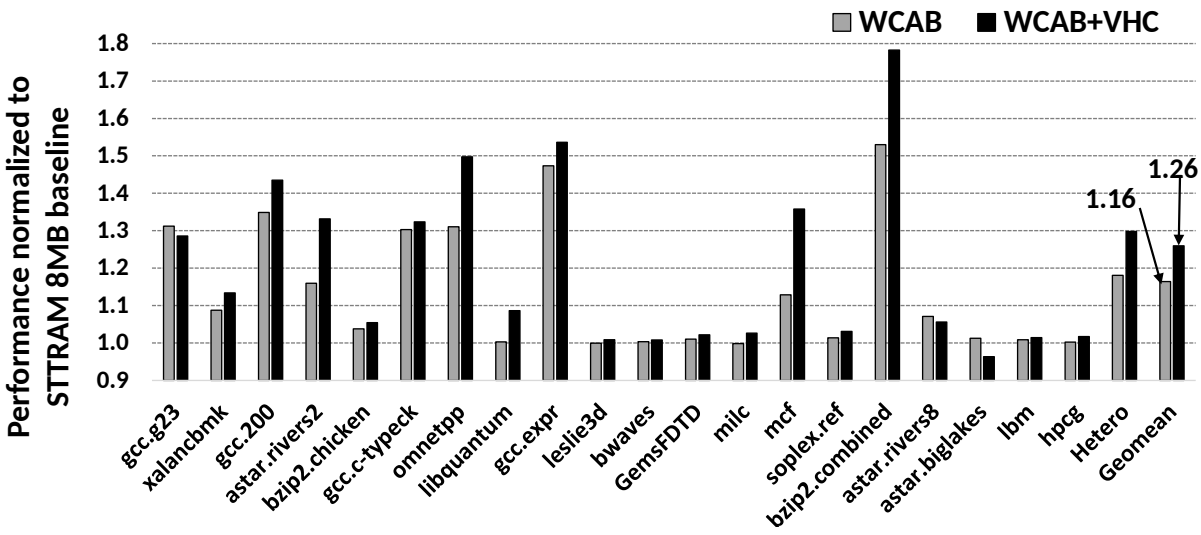


Figure 3.8. Performance in exclusive STTRAM LLC

analysis of our features, comparison with prior art, sensitivity to different parameters like write latency, banking, memory bandwidth and latency and a brief energy analysis. Further, we also present performance benefits of our techniques on a subset of industry workloads. Finally, we will compare different density and write latency options in STTRAM technology.

3.3.2.1 Baseline Bypass

We optimize the baseline LLC by deploying a PC based dead block predictor to bypass writes while improving the hit rate in the LLC. This policy is similar to the bypass policy proposed in DASCA [3]. This bypass predictor improves SRAM performance by 1% over an SRAM baseline that used the replacement and bypass schemes as proposed by [34]. We consider this predictor as prior art and employ it in the baseline of all the results in subsequent sections.

3.3.2.2 Performance Summary

(A) Exclusive LLC.

Figure 3.8 shows the performance gain of our proposal on top of an 8MB, 8 banks, exclusive STTRAM LLC baseline. Overall our proposal improves the performance (measured as

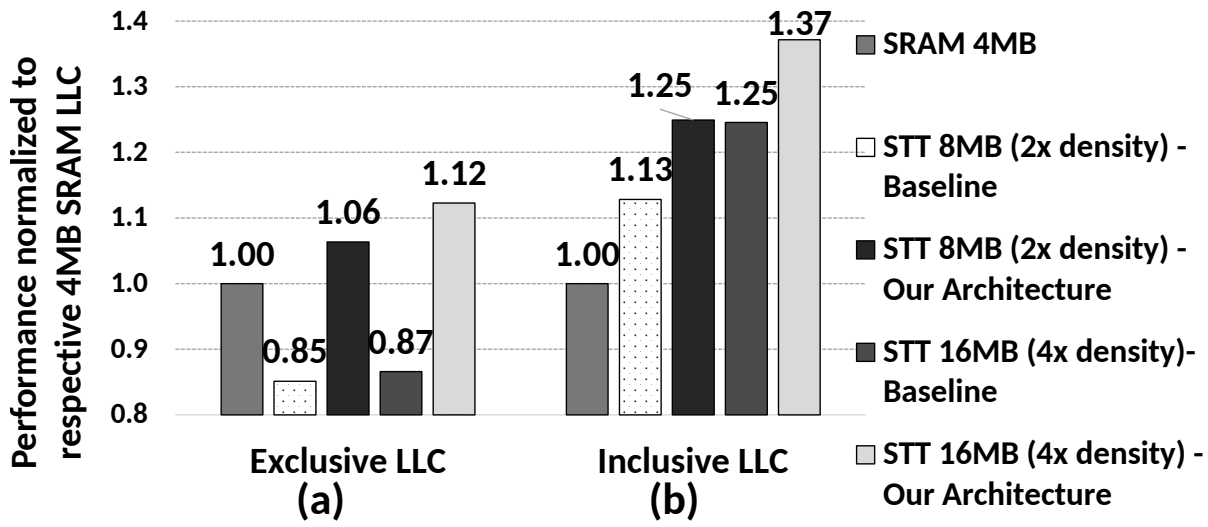


Figure 3.9. Performance compared to 4MB SRAM LLC

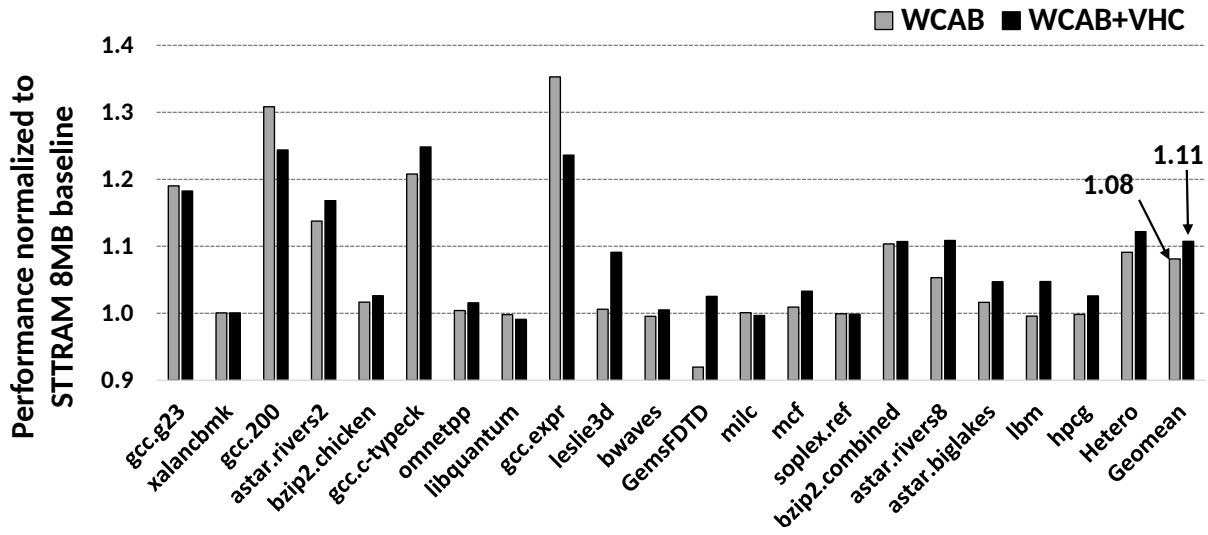


Figure 3.10. Performance in inclusive STTRAM LLC

geometric mean over all 64 traces) of the baseline STTRAM LLC design by 26%. The WCAB component of our policy gains 16% and VHC contributes additionally 10% more performance improvement. Several benchmarks gain significant performance. However hpcg, lbm and astar.biglakes do not show much response to our policies as these are streaming in nature and have very high LLC MPKI of 50, 28 and 27 respectively. Hence they are insensitive to LLC latency and bandwidth improvements.

Figure 3.9(a) compares our policies with a similar area SRAM. For these results we assume two density points for STTRAM in comparison to SRAM. Overall for exclusive LLCs we see that 8MB (2X density) and 16MB (4X density) STTRAM, despite of a larger capacity, lose 15% and 13% performance as compared to the 4MB SRAM LLC. However with our proposals added to the STTRAM LLC, we gain 6% and 12% performance in 8MB and 16MB STTRAM respectively.

(B) Inclusive LLC.

Figure 3.10 shows the performance gain of our features on top of an 8MB, 8 banks, inclusive STTRAM LLC baseline. For the inclusive baseline our features gain a significant 11% overall performance. We should note that our performance gains were expected to be lower in case of inclusive LLCs because inherently inclusive LLCs have less write pressure as the clean writebacks from the L2 are simply dropped.

Figure 3.9(b) compares our policies with a similar area SRAM. As described earlier, inclusive LLCs see fewer writes, therefore the baseline STTRAM with 8MB and 16MB provide 13% and 25% performance gains. Our proposals further improve performance to 25% and 37% for 8MB and 16MB STTRAM respectively.

We now analyze our policies in more detail in subsequent sections. For sake of brevity, we will show results only on the exclusive LLC baseline.

3.3.2.3 Performance Analysis

(A) Write Congestion Aware Bypass.

Figure 3.11 shows the gains delivered by WCAB for every trace used in our simulation (left Y axis). Also plotted is the miss rate difference from this baseline on the right Y axis. On an average, WCAB reduces the writes to the LLC by 11%, while increasing the miss rate by 9%. This trade-off helps it deliver performance. We should note that there are some benchmarks that lose because of WCAB. These benchmarks typically suffer a higher miss rate that is not offset by the improved congestion at the LLC. Overall WCAB performance ranges from -3% to +53%.

To verify the significance of PC based liveness, we changed WCAB bypass algorithm to not use liveness score. Instead it selects the oldest write for bypass when there is congestion. This scheme degrades the performance by 3% compared to our default WCAB algorithm using PC based liveness.

(B) Virtual Hybrid Cache.

VHC further improves the performance gain by 10%. Figure 3.12 shows the gains delivered by VHC on top of WCAB, for every trace. Also plotted is the difference in miss rate. Overall VHC reduces the number of writes to the LLC and hence helps reduce the aggressiveness of WCAB, thereby improving the overall hit rate. Some losses seen with VHC are primarily because of additional hit rate loss as a result of duplication (that reduced the effective capacity of LLC + L2). VHC is able to eliminate nearly 40% of writes at the LLC, while increasing the miss rate further by 2.2%.

3.3.2.4 Comparison to Prior Art

We compare our architecture to two existing policies namely the Hybrid Cache (as used in [115]) and LAP [17]. These two policies were discussed (in related works). Figure 3.13 compares the proposed architecture with these two existing policies. For the hybrid cache we used two configurations where the SRAM portion is 1/4 th and 1/8 th of the LLC capacity. If

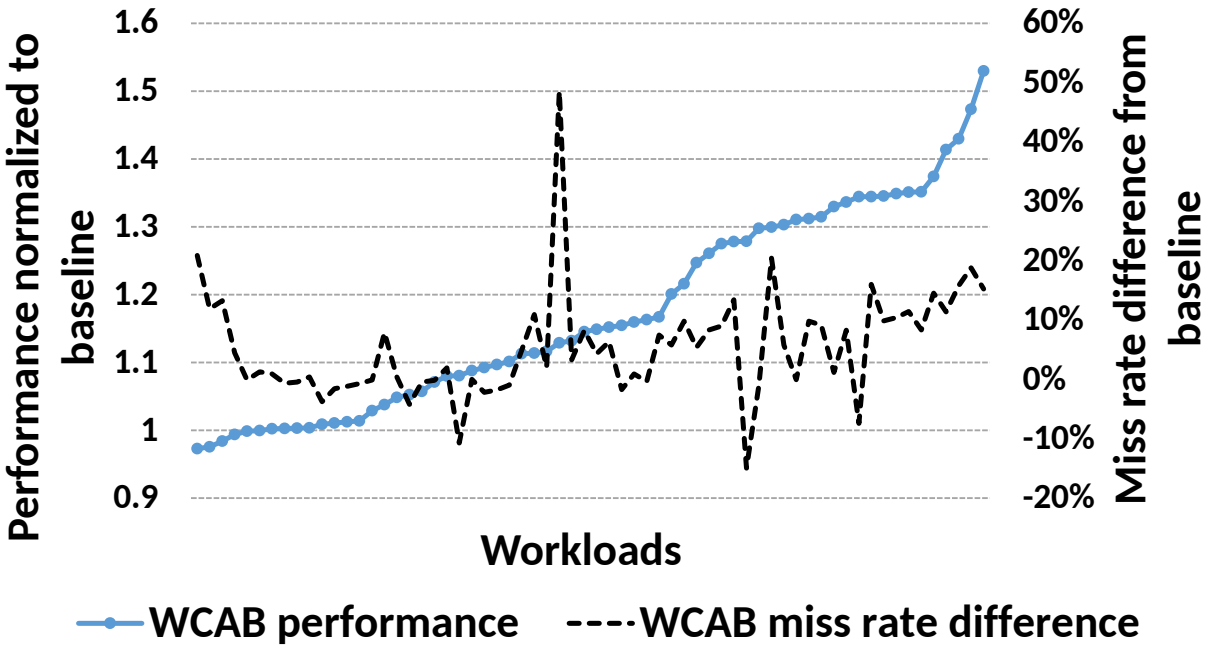


Figure 3.11. Performance vs. miss rate difference of WCAB

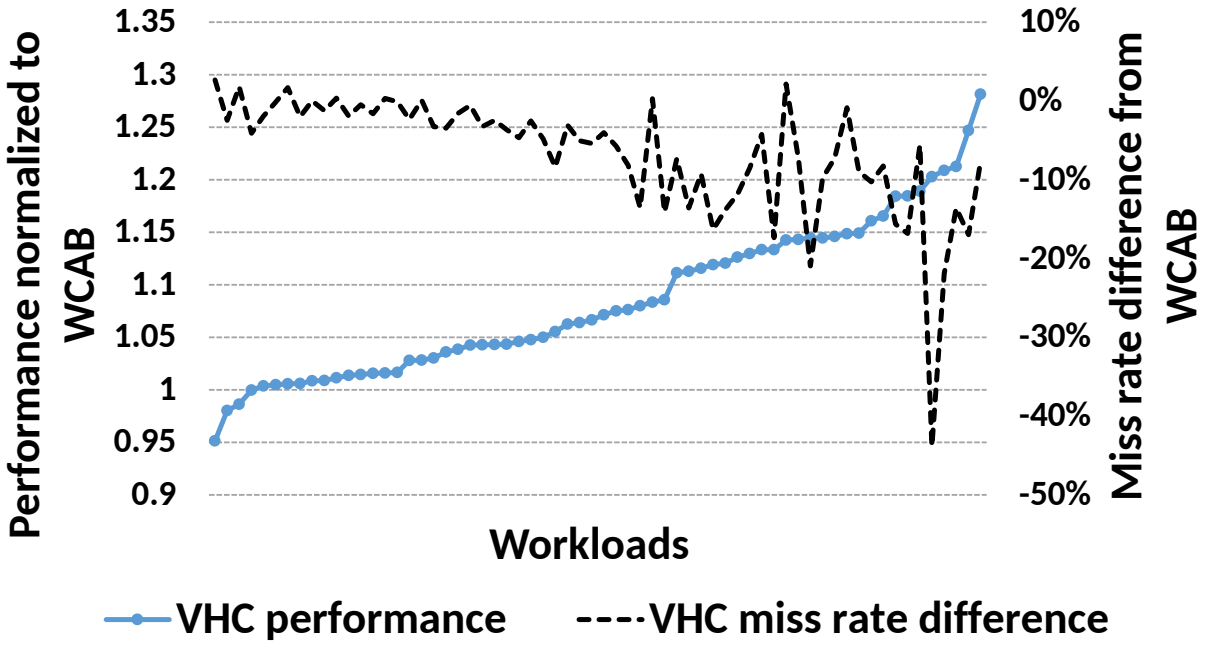


Figure 3.12. Performance vs. miss rate difference of VHC on top of WCAB

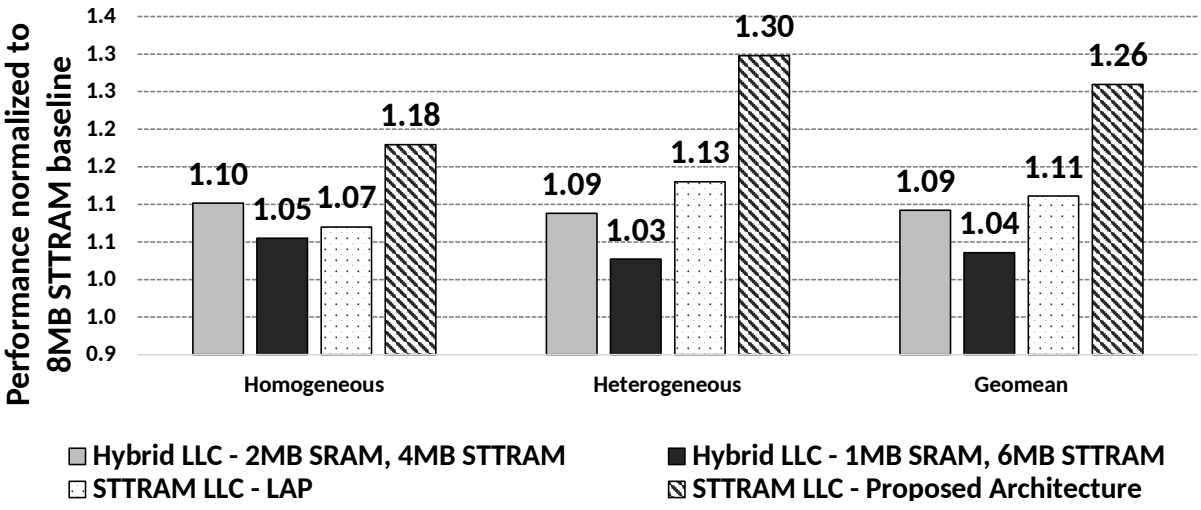


Figure 3.13. Performance comparison to prior art

we assume STTRAM has 2x density as com- pared to SRAM, the two configurations will have have 4MB and 6MB of STTRAM respectively. On an average the best performing Hybrid cache improves the performance by 9%, whereas LAP improves the performance by 11%. On the other hand, the proposed architecture improves the average performance by 26%.

As discussed earlier, Hybrid caches result in an over- all smaller capacity, as SRAM has a lower density than STTRAM. Our simulations show that the best performing Hybrid cache (with 2MB of SRAM and 4MB of STTRAM) has an average 11% higher miss rate, as compared to the STTRAM baseline. On the other hand the fraction of writes drop by 19%. The capacity loss tends to bring down the gains of Hybrid cache.

LAP shows a performance improvement of 11%. Since cache lines are duplicated by LAP there is a loss in cache capacity. To overcome this capacity loss, the proposed Virtual Hybrid Cache uses intelligent replacement optimizations as described (under Virtual Hybird Cache). Moreover, unlike LAP that cannot target dirty fills, VHC uses a small subset of the L2 cache to increase write merging and cut down on dirty fills. To show the advantages of the proposed optimizations by VHC we compare just the VHC component of our policy with LAP in Figure 3.14. As can be seen for several traces, VHC has a lower miss rate loss as compared to

LAP.

As a standalone feature itself, VHC outperforms LAP by 2.6% on the same baseline. VHC has a 1.5% lower miss rate as compared to LAP, though LAP has 2.8% lower writes to the LLC. Overall our proposed architecture delivers 26% performance as compared to LAP that delivers 11% performance and Hybrid cache that delivers 9% performance gain for an 8MB STTRAM, having a 20 ns write latency.

3.3.2.5 Sensitivity Studies

(A) Sensitivity to LLC Banking.

Banking the LLC can increase the overall LLC bandwidth and help mitigate the LLC congestion. However banking results in substantial area and power overheads because of duplication of peripheral circuit, interconnect etc. [81]. Also floorplan considerations are important to decide the number of LLC banks that are possible. Figure 3.15 shows the impact of our policy when the number of banks are increased in the baseline. For 8 banks, we gain 26% performance. With 16 banks (and assuming no area overheads of banking), the STTRAM baseline improves by 18%, and our policies improve the gain further by 15%. However banking costs area. A 10% overhead reduces the performance gain of 16 banks over 8 banks to 16%, whereas a 20% overhead would further reduce the performance gain to 15%. Our policies gain 12% and 11% performance on top of these area compensated banked baselines.

(B) Sensitivity to Write Latency.

Figure 3.16 shows the performance of our policies at different write latencies of an 8MB STTRAM LLC. Our policies gain 1%, 7%, 26% and 53% performance gains at STTRAM LLC write latency of 5ns, 10ns, 20ns and 40ns respectively. Write congestion problem is less severe at lower STTRAM write latencies and hence the sensitivity of our proposals increase as the write latency of the STTRAM is increased. However we should note that reducing write latency at the circuit level reduces the density of the STTRAM, and may not be desirable. We will evaluate

this interesting trade-off in a separate section (STTRAM density versus Write Latency).

Figure 3.16 also shows the performance gain of our policies on an 8MB SRAM (+0ns write latency), where there is no write congestion. As expected our policies overall degrade the performance of such a baseline by almost 2%. This shows the uniqueness of our problem space and our solution - conventional SRAM LLCs expectedly lose performance when we degrade hit rates, whereas for STTRAM LLCs we can actually lose some amount of hit rates as long as we can use that loss in order to improve LLC characteristics and gain overall higher performance.

(C) Sensitivity to Algorithmic Parameters.

Our thresholds are dependent on the memory latency (the cost of bypass) and the write latency of the STTRAM LLC. Therefore these need to be tuned for a given system. We ran several experiments to arrive at the best possible thresholds. For our default system configuration the best thresholds are mentioned in Figure 3.6.

(D) Sensitivity to Last Level Cache Capacity.

Figure 3.17 shows the performance delivered by our optimization at various LLC sizes by keeping the write latency constant at 20ns. For each capacity point we also show the performance that an SRAM would deliver at the same capacity. Note that SRAM would need a much higher area than STTRAM for a given capacity, so this comparison is primarily to gauge the effectiveness of our policy in mitigating write latency. Overall we see that a similar capacity SRAM would deliver 20%, 36% and 42% performance on top of 4 MB, 8 MB and 16 MB STTRAM baseline respectively. However, with our architecture applied to the baseline STTRAM, we bridge this gap by almost 65-70% across the various capacity points. The remaining gap between SRAM and STTRAM can potentially be bridged by improving the bypass decisions of WCAB so that capacity is not sacrificed while writes are still bypassed. Also addressing dirty fills and write merging in the core can significantly bridge this gap.

(E) Sensitivity to Memory Bandwidth and Latency.

We evaluated different main memory bandwidth and latency configurations with an 8

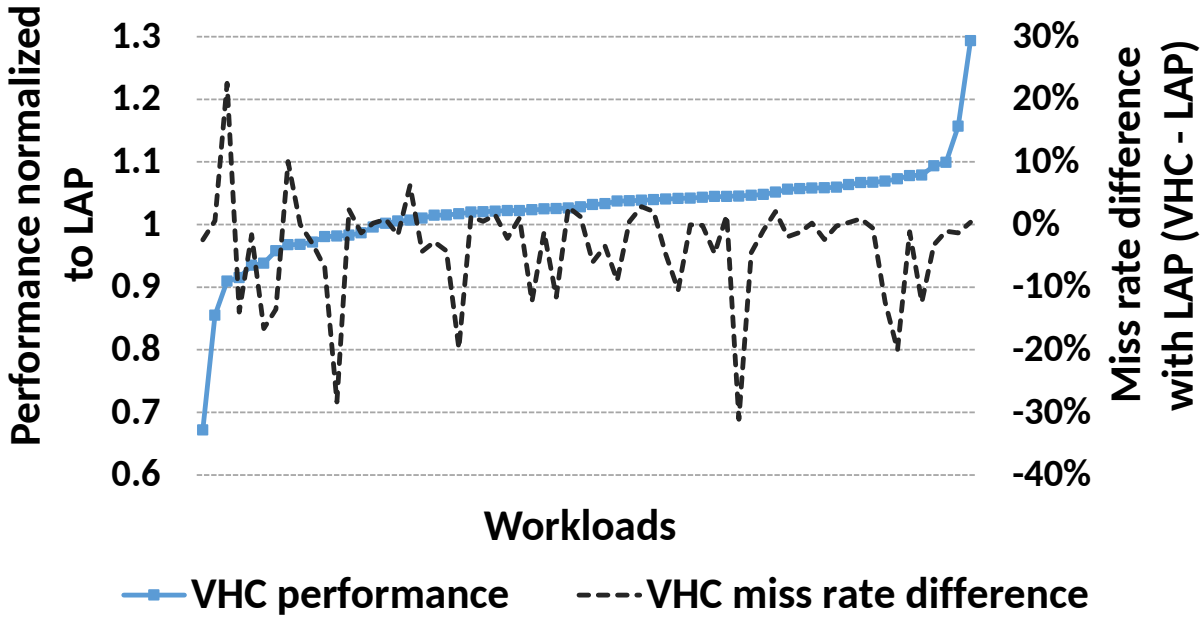


Figure 3.14. Performance and miss rate difference of VHC compared to LAP

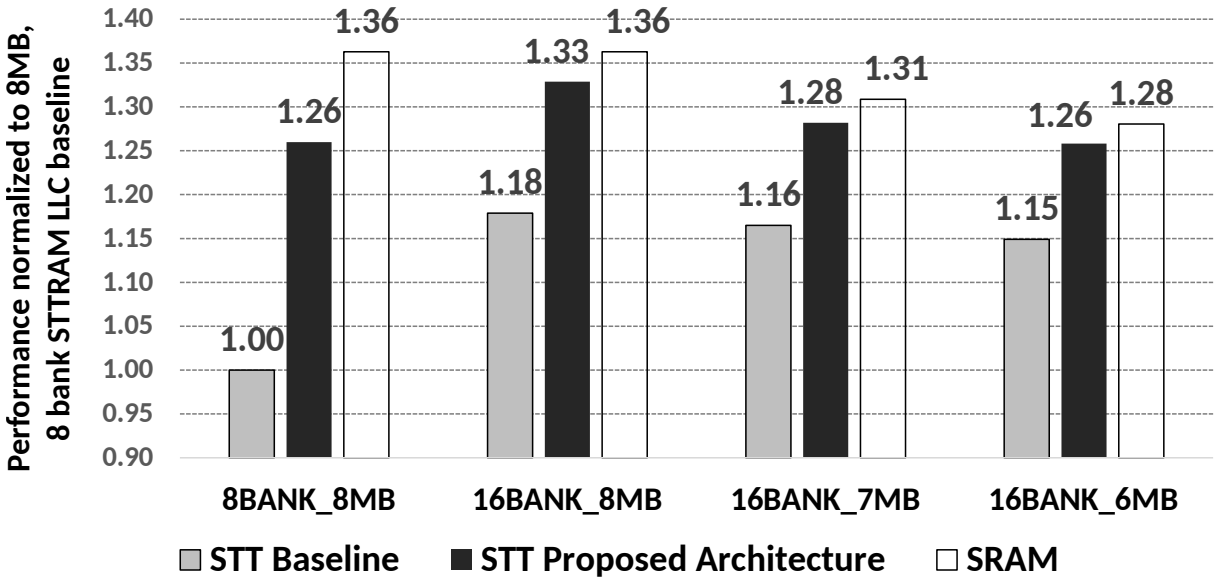


Figure 3.15. Impact of LLC banking

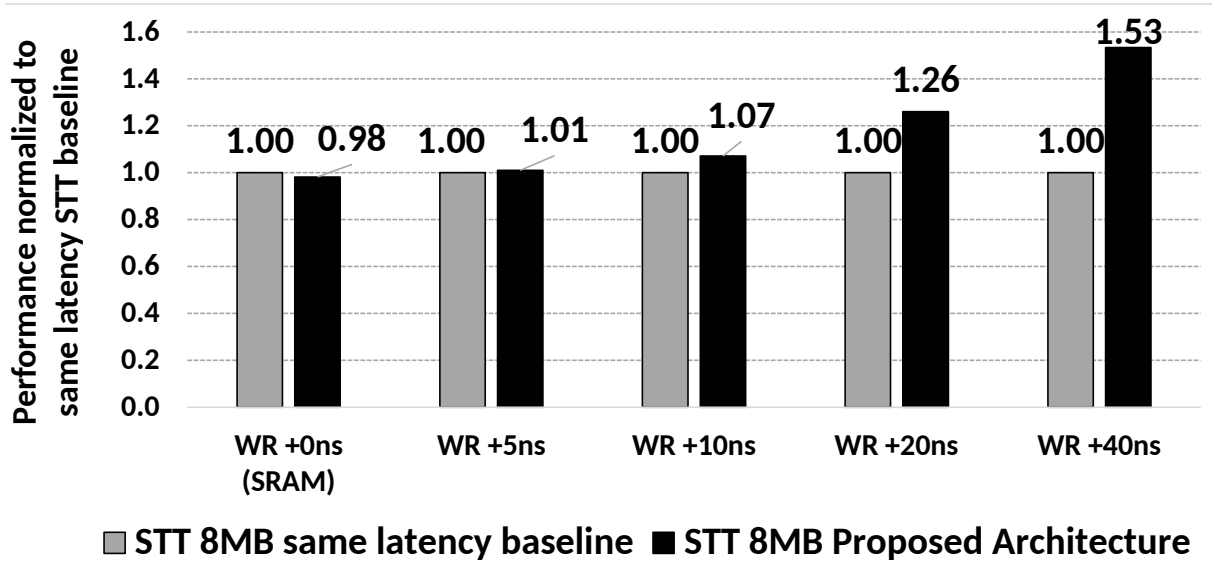


Figure 3.16. Impact of STTRAM write latency

MB STTRAM LLC baseline. Compared to the default DDR-2400 memory, using DDR-1600 lowers performance by 2%, whereas higher bandwidth DDR-3200 improves by 1%. On the other hand increasing memory latency by 50% reduces performance by 4%, whereas reducing latency by 30% increases performance by 5%.

Figure 3.18 shows the performance delivered by our proposal at different memory bandwidth and latency points. Expectedly our performance gains increase when the baseline memory bandwidth is increased. This is because higher bandwidth memory will be able to absorb the extra pressure our write bypassing scheme creates at the memory. Our proposal gains 27% when the baseline uses a higher bandwidth DDR4-3200, whereas the gains are somewhat lower at 24% for lower bandwidth DDR4-1600. Likewise better latency memory baseline improves our gains to 29%, whereas high memory latency baseline reduces our gains to 22%. In summary, our proposals deliver substantial performance at different memory configurations.

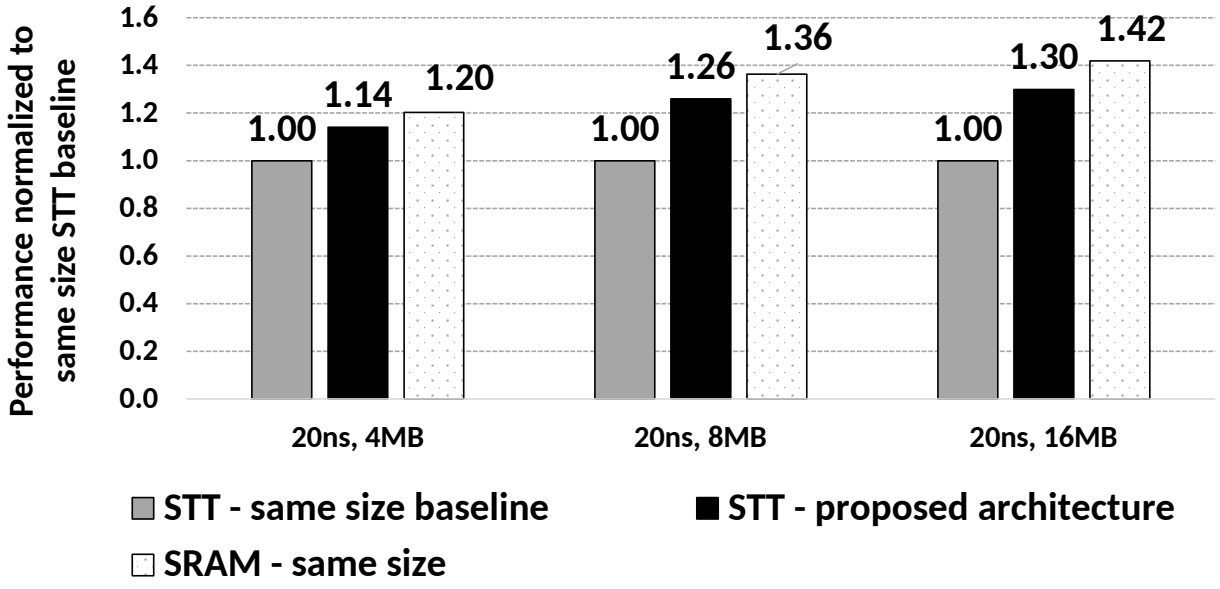


Figure 3.17. Our techniques vs same capacity SRAM

3.3.2.6 Energy Considerations

STTRAM has high write energy requirements [17, 3]. Our proposed architecture reduces this write energy by eliminating writes. However, this comes at a higher miss rate which costs power in the off-chip DRAM memory. To estimate the various components of energy in the proposed architecture, we use the Micron power calculator [47] for estimating the DRAM array energy in the main memory. For STTRAM read/write energy we use the same numbers as used in [17]. Figure 3.19 shows the LLC and main memory energy of our proposed architecture, normalized to the baseline STTRAM. Our architecture, by eliminating writes to the STTRAM, reduces the energy across all workloads by an average of 8%. However in some workloads (for instance gcc.g23, gcc.expr) the energy increases due to additional bypasses that lower the hit rate and result in more requests at the DRAM memory.

3.3.2.7 Industry Workloads

Apart from the workloads described earlier, we evaluated our proposal on 15 important industry applications from Enterprise, Database, BigData, Desktops, Mobile and HPC domains.

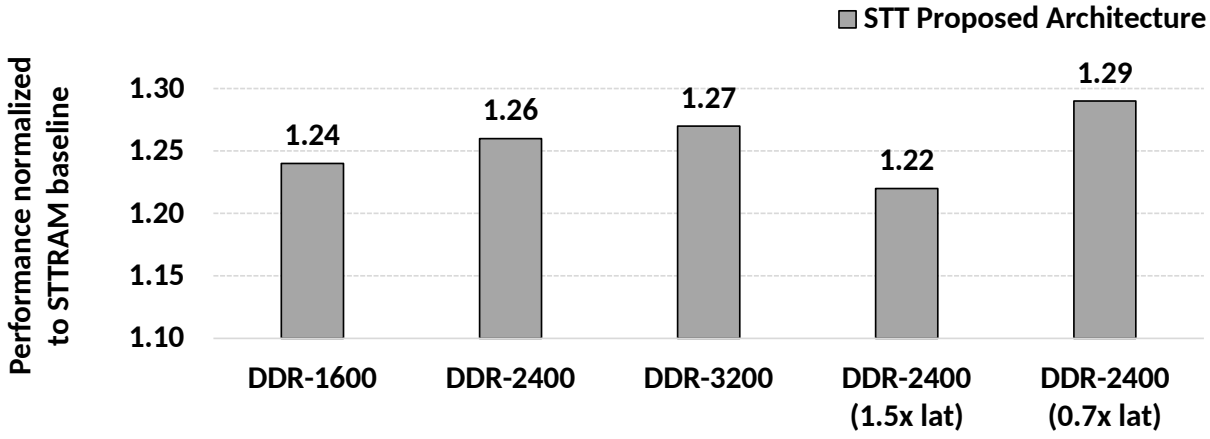


Figure 3.18. Impact of memory bandwidth and latency

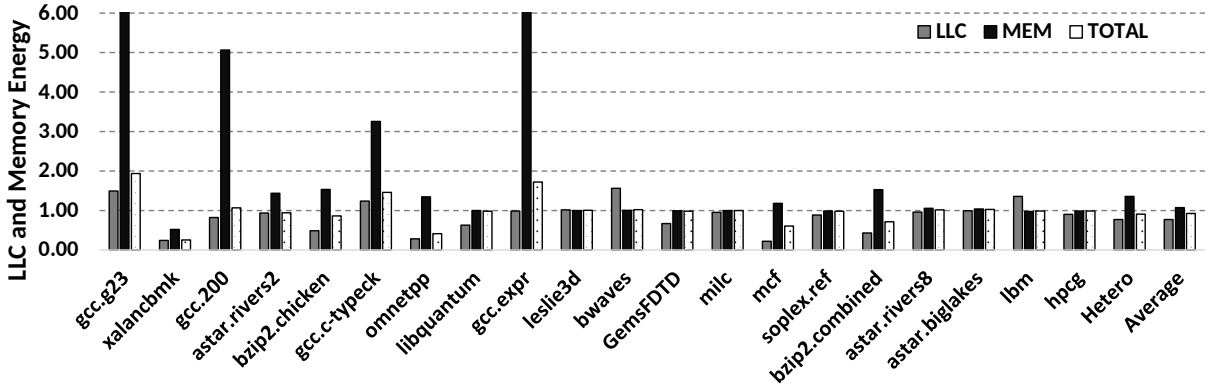


Figure 3.19. Combined LLC and main memory energy

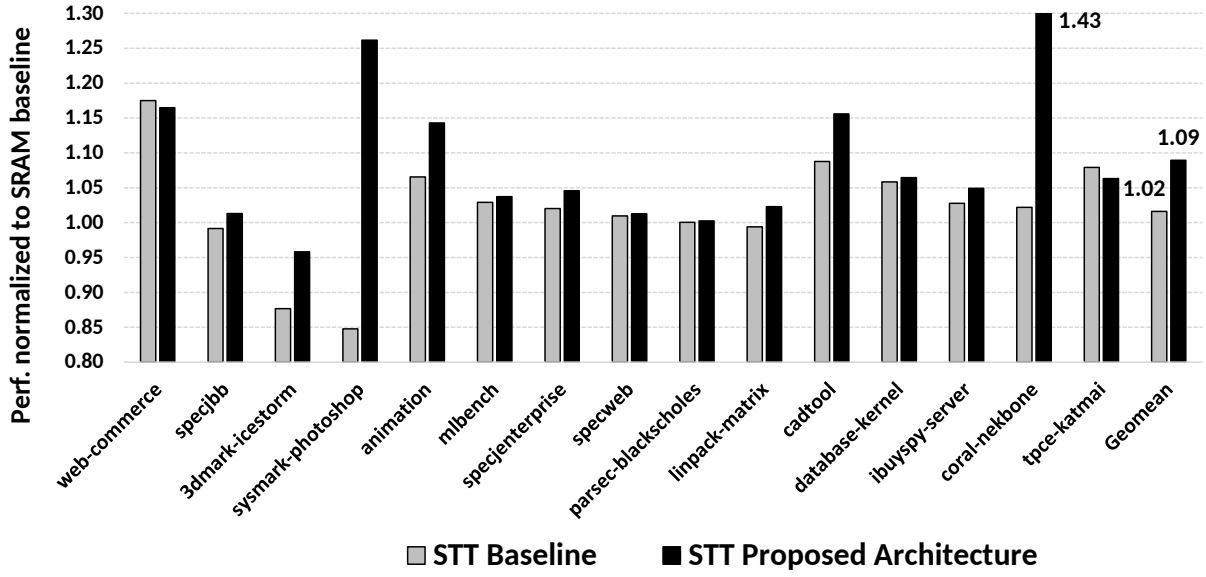


Figure 3.20. Evaluation on industry workloads

Table 3.2. STTRAM write latency vs cell density

Write Latency	5ns	10ns	20ns	30ns
Conservative density	4MB	6MB	8MB	12MB
Aggressive density	7MB	12MB	16MB	20MB

Most of these applications run on real world commercial systems and some of them represent emerging usage like machine learning. Moreover, these workloads exhibit a wide range of characteristics such as LLC MPKI and write intensity, stressing different tradeoffs in the cache hierarchy. Figure 3.20 shows the performance of STTRAM baseline and with our optimization on these industry workloads compared to the SRAM baseline. Overall, STTRAM with our optimization provides 9% performance advantage over the SRAM baseline showing that our proposal is applicable on real world workloads with broad range of characteristics.

3.3.2.8 STTRAM Density versus Write Latency

As discussed earlier, write latency determines the density for STTRAM technology. Higher write latency results in higher density and vice versa. We consider two different density

scaling characteristics based on the data available from [57] and summarized in Table 3.2. The two capacity-latency curves respectively represent an aggressive and conservative density scaling STTRAM.

Figure 3.21 shows the performance with conservative and aggressive capacity scaling as the write latency increases from 5 ns to 30 ns. In conservative scaling (Figure 3.21(a)), the baseline STTRAM shows performance loss and the loss increases further with the write latency. On the other hand, our proposed architecture is able to mitigate higher write latency and take advantage of larger capacity, thereby delivers 6% performance gain at (20ns, 8MB) compared to same area SRAM. Performance advantage of our architecture further increases to 18% when applied to aggressive density scaling as shown in Figure 3.21(b). Moreover, note that our proposed architecture sees a more gradual decline in performance for (20ns, 16MB), and (30ns, 20MB), when compared to baseline which falls off a cliff.

The results show that in presence of our optimizations, the best performing write latency is 20ns for conservative and 10ns for aggressive scaling. Without our optimizations, circuit techniques would need to reduce latency further to 10ns and 5ns for conservative and aggressive scaling configurations, thereby sacrificing significant LLC capacity. This clearly shows that a combination of circuit techniques to reduce write latency, coupled with our proposed architecture techniques, helps create a more optimal NVM LLC design.

3.4 Related Work

A large body of work [15, 56, 16, 34, 51] has addressed the capacity management problem in SRAM based LLCs. Managing the demand for LLC bandwidth has been addressed in the context of SRAM LLC [102, 34] mainly to reduce on-chip traffic and power. NVM LLCs can also benefit from these techniques. However, NVM has long write latency and therefore needs solutions to mitigate the LLC congestion arising because of these long latency writes. Some of the solutions explored in the past to this problem include [109, 114, 115, 17, 3, 54, 124].

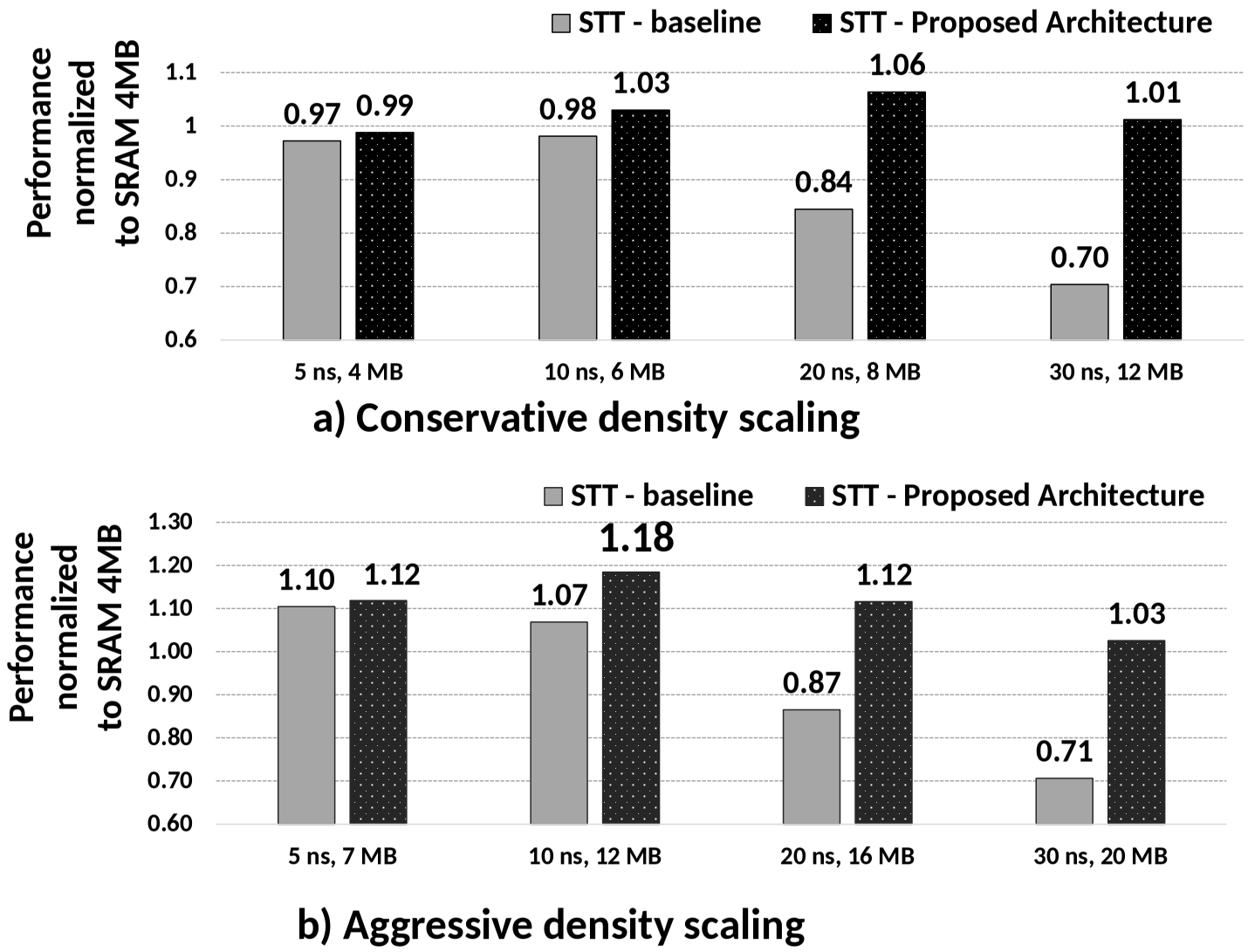


Figure 3.21. STTRAM density vs write latency scaling

These solutions propose heuristics to either stall writes or completely bypass the writes. These techniques eliminate some of the writes while making sure there is no loss in hit rate because of the bypass. Bypassing writes helps relieve some of the LLC congestion. However increasing the aggressiveness of bypass to improve LLC congestion further is very risky as it can reduce hit rates significantly and severely degrade performance.

Hybrid cache [115] proposes to convert some portion of the NVM LLC into an SRAM. Frequently written to cache lines are allocated in the SRAM portion to reduce writes to the primary NVM portion. However, as SRAM has lower density than NVM, this results in overall lower capacity than a pure NVM based LLC and hence bounds the performance of capacity sensitive applications. The Hybrid cache proposal uses a predictor for placement and migration between the SRAM portion and NVM to reduce write congestion.

LAP [17] was proposed for exclusive STTRAM LLCs. It duplicates cache lines, that make frequent trips between the LLC and the L2, in the LLC. However, this duplication results in LLC capacity loss limiting the performance potential. Moreover a significant fraction of writes to the LLC originate from dirty writebacks from the L2 which cannot be addressed by this scheme. We have compared the Hybrid cache proposal and LAP with our architecture in the results section.

Optimizations have also been explored in the context of memories built using NVM technology by [124, 63, 74]. The focus of these works is primarily to improve energy efficiency and resilience of NVM memory. FIRM [125] proposes to reduce memory bus turnarounds and utilizes bank level parallelism to improve fairness and performance in NVM based persistent main memory systems. A recent work, OSCAR [123], tries to solve high bandwidth demand on on-chip network of shared GPU-CPU NVM based LLC.

Overall, although past efforts have shown improvement over baseline NVM LLCs, none have explored techniques that delicately balance the conflicting goals of managing contention while retaining the benefits of higher capacity. Through a comprehensive and synergistic set of architectural techniques we show a path to get to performance that is within striking distance of

that of an LLC with SRAM-like write latency and of the same capacity.

3.5 Conclusion

In this paper we showed that LLCs built with emerging NVM memory technologies like STTRAM give sub-optimal performance as compared to SRAM because of long write latency. We hence proposed a new, low cost, architecture that mitigates this write latency induced performance degradation and improves the performance of a 4 core system with an 8MB of STTRAM based exclusive LLC by an average of 26%. Moreover, we show that the proposed architecture can tolerate high asymmetry in write latency and delivers significant performance improvements over a traditional SRAM LLC. This can pave the path for the creation of future large LLCs that can effectively utilize the high density offered by these new NVM technologies, while still delivering near SRAM-like performance.

3.6 Acknowledgement

Chapter 3, in most parts, has been published in the Proceedings of the *45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 315-327; Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache; Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young and Hong Wang, 2018. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Scalability in In-Memory Computations

Scaling limits of non Von-Neumann computing models

Processing huge amounts of data on traditional von Neumann architectures involves many data transfers between the CPU and the memory that degrades performance and consumes energy [90, 95, 96]. Enabled by emerging memory technologies, processing in memory (PIM) is one solution that might reduce the costly data transfers. With true PIM, computations can be performed using the individual memory cells [91, 118, 71, 64, 30]. Research on PIM in recent years has led to better circuits and micro-architectures [64, 65, 9, 30] and identification of applications which can take advantage of this paradigm [46, 26, 37].

Despite the recent resurgence of PIM, it is still very challenging to analyze and quantify its advantages or disadvantages over other computing paradigms. Simple heuristic based mapping (such as "only map simple operations to PIM") may not fully capture the capability of PIM, but mapping complex operations without a clear view of PIM's limits may not be an appropriate approach either.

In this paper, we propose an analytical model – the Bitlet model – to address the challenge of better understanding PIM. This model is inspired by past successful analytical models for computing [36, 39, 117, 27, 40]. The model provides a simple operational view of PIM computations, it abstracts PIM implementation details, and it captures essential factors such as complexity of operations and the extent of data transfers. The Bitlet model exposes clear trade-offs between the different computing options.

The Bitlet model provides sufficient depth of detail to include various parameters related

to technology, architecture and algorithms which affect the performance and power of PIM based processing. The name Bitlet reflects PIM's unique bit-by-bit data element processing approach. Bitlet is complementary to past CPU/GPU models focused solely on arithmetic intensity or data-reuse [117].

Overall, we make the following contributions:

- Our analytical model quantifies the effects of computational complexity and data transfer factors on PIM performance.
- Our model also permits a fair, parameterized throughput-centric comparison between PIM and traditional CPU/GPU computing units.

Throughout the paper, we refer to 'PIM' as a framework for processing inside memories. We ground the PIM side of the Bitlet model on the concept of performing computations using memristive memory arrays. PIM processing occurs inside the memory arrays and at the level of individual memory cells. We base our model of PIM on the MAGIC in-memory logic family [64]. The supporting circuitry and micro-architecture resemble, but are not limited to, those described for the memristor Memory Processing Unit (mMPU) [44, 38].

4.1 Bitlet Model

We derive a parameterized throughput metric when the data is processed by PIM, followed by a suitable throughput metric for the CPU.

4.1.1 PIM Throughput

In our proposed Bitlet model for PIM, we assume that the computations are carried out as a series of NOR operations, applied on the memory cells of a row inside a memristive memory array. In the model, each row of a memory array stores the input data required for processing. A two-input bit NOR gate processes two data bits within the row and stores the

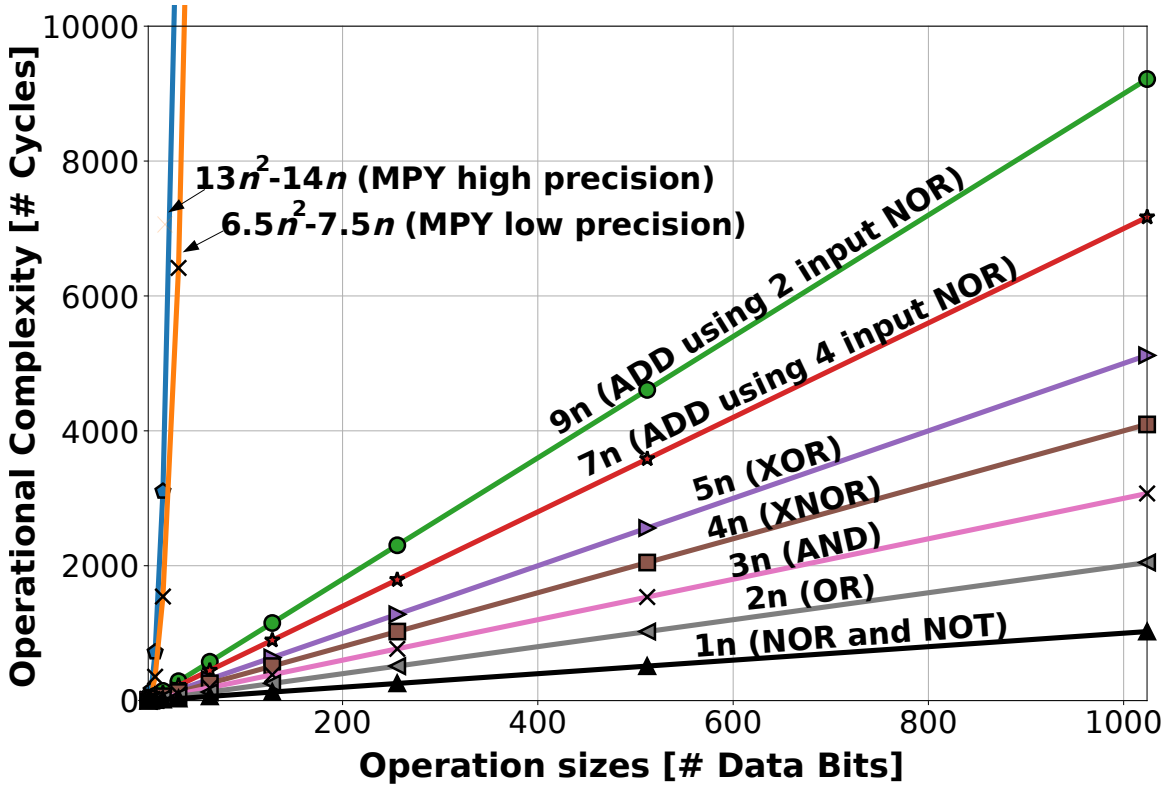


Figure 4.1. PIM operational complexity in cycles for different types of operation and data sizes. MPY refers to a multiplication operation. Other arithmetic and logic operations are also shown.

output bit in the same row. Any intermediate data is processed similarly. Processing proceeds sequentially in this fashion to produce the final output, which is also stored within the same row. The processing of data as per the Bitlet model can thus be viewed as *row-wise* and *bit-by-bit* within the row of a memory array. We use a default two-input bit NOR gate as the basic logic operation [64], permitting a maximum of two input bits to be processed per memory cycle, unless stated otherwise.

While each row is processed bit-by-bit, the effective throughput of PIM is increased by the inherent parallelism: multiple rows are processed inside a memory array and multiple memory arrays are available, all operating concurrently. We assume the same computations (*i.e.*, individual operations) applied to a row are also applied in parallel in every cycle across all the rows (*ROW*) of a memory array. This parallelism is made possible by the 2D structure of

Table 4.1. PIM-related parameters of the Bitlet model.

Parameter name	Notation	Value(s)	Type
Operation complexity	OC	1 - 32k cycles	Algorithmic
PIM cycle time	CT	10 ns [68]	Technological
Memory array count	MAT	1k - 16k	Architectural
Memory array dimensions	$ROW \times COL$	1024 x 1024	Technological

the memory arrays and by reusing the voltage signals used to operate an individual row for all the rows [44]. Although the choice to process row-wise only may seem restrictive, it naturally maximizes the data-level parallelism and hence PIM throughput. Moreover, the multiple memory arrays (MAT) further maximize this parallelism. Finally, the cycle time, CT, of a single basic PIM operation also impacts overall PIM performance. The shorter the cycle time, the faster the processing.

Figure 4.1 shows how bit lengths (n) of the input data affect the number of computing cycles required for PIM based processing. The figure shows the number of cycles needed to process the input data and produce the desired output under the row-wise and bit-by-bit processing model and highlights how this time is affected by both the data sizes and type of operations (different operations follow a different curve on the graph). With this model, for example, n-bit AND requires $3n$ cycles (*e.g.*, for n=16 bits AND takes 16x3 = 48 cycles), ADD requires $9n$ cycles[1], and multiply (MPY) requires $13n^2$- $14n$ cycles [37]. We define the **operational complexity** parameter (OC), for a given operation type and data size, as the number of cycles required to process the corresponding data.

The throughput of PIM is captured by four parameters: OC, MAT, ROW and CT. The throughput of the system in operations per second can be expressed as:

$$Throughput\text{-}PIM(Op) = \frac{ROW \times MAT}{OC \times CT}. \tag{4.1}$$

Table 4.1 summarizes the PIM-related parameters of the Bitlet model. For conceptual

[1] ADD can be improved to $7n$ cycles using an algorithmic optimization that uses four-input NOR instead of two-input NOR.

Table 4.2. CPU-related parameters of the Bitlet model.

Parameter name	Notation	Value(s)	Type
Memory bandwidth	*BW*	1T to 16T (T = Tbps)	Architectural
Data in-out bits	*DIO*	24, 48	Algorithmic

clarity and to aid our analysis, we designate three parameter types: *technological*, *architectural*, and *algorithmic*. Typical values or the ranges for the different parameters are also listed in the table.

Examples: The examples below illustrate the throughput of PIM computed as per the Bitlet model.

ADD 16-bit, OC=144: Consider an ADD operation which adds two 16-bit inputs and produces a 16-bit output. This operation on a data element takes 144 cycles (OC = 144, $9n$ where n=16). Assuming there are 1024 MATs and each MAT supports 1024 data elements (rows= # data elements), the achieved throughput = (1024x1024)/(144x10) = **728** GOPS.

OR 16-bit, OC=32: Consider a 16-bit OR operation which ORs two 16-bit inputs and produces a 16-bit output. In this case, OC = 32 ($2n$ where n=16) and the throughput = (1024x1024)/(32x10) = **3276** GOPS.

MPY 16-bit, OC=3104: Now consider a 16-bit MPY (multiplication) producing a 32-bit result. In this case OC = 3104 ($13n^2$- $14n$ where n=16). Here the throughput is (1024x1024)/(3104x10) = **33** GOPS. For low-precision multiplication that produces a 16-bit output, OC = 1544 and the throughput is (1024x1024)/(1544x10) = **67** GOPS.

4.1.2 CPU Throughput

We now discuss the CPU and the associated assumptions we made as per the Bitlet model.

Given the focus of the Bitlet model on exploring compelling cases for PIM, we assume in this setting that the CPU execution speed is limited by its effective usage of external memory-bandwidth and not by the processing speed of its execution units. In this way, we distinguish

between PIM and CPU processing: performing all computations inside the memory arrays without data transfer, versus transferring data when performing any computation. Large amounts of data being transferred between the CPU and the memory will result in lower throughput while less data will have the opposite effect.

When the operation is performed by the CPU, the extent of data transfer between the CPU and the memory is captured by the **data in-out** (DIO) model parameter. The DIO is the amount of data transferred on average per operation, and has to account for all the data transfers (in bits) between the CPU and the memory due to inputs, outputs and any temporary results. Along with DIO, the external memory bandwidth (BW) between the CPU and the memory[2], determines the final throughput.

The throughput of the system in operations per second can be stated as:

$$Throughput\text{-}CPU(Op) = \frac{BW}{DIO}. \tag{4.2}$$

Table 4.2 lists the CPU-related parameters, including typical values or range of values they are set to. We vary the memory bandwidth parameter from 1 Tbps to 16 Tbps to show sensitivity of the model to memory bandwidth.

Examples: We now discuss a few examples of using the Bitlet model for CPUs. Here we vary the memory bandwidth parameter to a few values to illustrate how the model works.

'X' Op 16-bit, BW = 4 Tbps: We consider any binary operation ('X' stands for ADD, OR, MPY, etc.) that operates on two 16-bit inputs and one 16-bit output. The DIO is thus (16x2+16) = 48 bits[3]. For any of these operations, the effective throughput of the CPU is 4T/48 = **85** GOPS. In comparison with CPU throughput for OR and AND in the earlier examples, PIM throughput is better primarily due to lower operational complexity, high data parallelism, and (unlike the CPU) by virtue of being unaffected by external memory bandwidth. For MPY, on the other hand, PIM is inferior to the CPU due to the higher operational complexity.

[2]Memory bandwidth may depend on the number of channels.

[3]DIO = 24 for two 8-bit inputs and one 8-bit output.

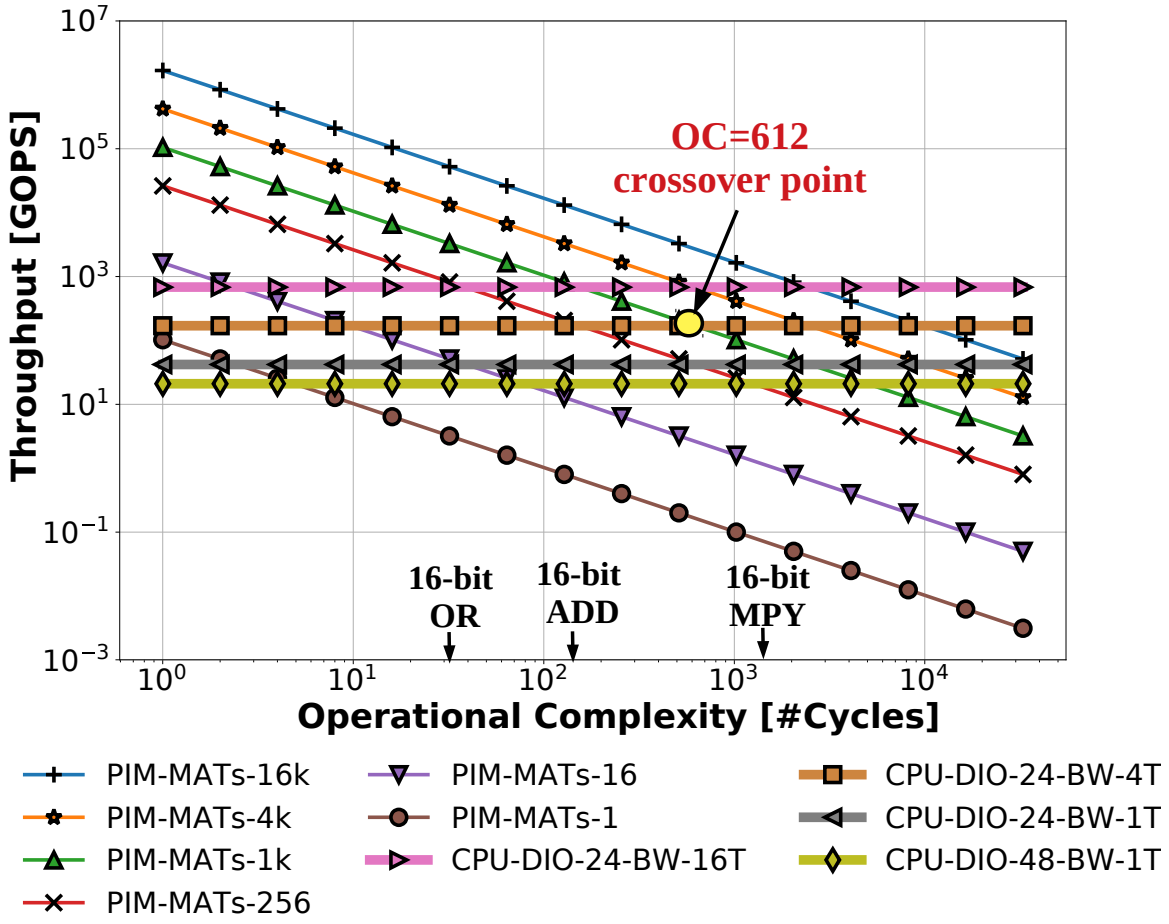


Figure 4.2. Throughput comparison of CPU vs. PIM. A crossover point where the CPU starts performing better than PIM is shown. The crossover point is for a particular PIM, CPU configuration

'X' Op 16-bit, BW = 1 Tbps: If the bandwidth is reduced to 1024 Gbps, the throughput now becomes 1024T/48 = **21** GOPS for any 16-bit binary operation with 16-bit output. Since memory bandwidth is the main limiter, CPU throughput now becomes worse than PIM, even with respect to the MPY operation.

4.1.3 PIM versus CPU Comparison

We start with the raw PIM vs. CPU throughput comparison and then study how power limitations impact the results.

4.1.3.1 Comparison of Raw Throughput

Figure 4.2 shows the throughput of PIM against that of the CPU. Diagonal lines represent PIM with varying numbers of $MATs$ (set to 1, 16, 256, 1024, 4096 and 16384 $MATs$). Horizontal lines are for CPUs with varying DIO bits (set to 24/96) and BW = 1T/4T/16T (T = Tbps).

Peak throughput for PIM occurs when maximum available memory arrays are used ($MATs$ = 16k) and the operational complexity is the lowest possible (OC = 1). We observe that PIM throughput increases when more MATs are used and decreases with increasing operational complexity. We can see that the CPU throughput decreases with higher DIO. For instance, consider the lines shown for DIO = 24 and DIO = 48 for the same BW = 1T. The CPU's performance for DIO = 48 is lower than for DIO = 24.

For a configuration of MAT = 1024, DIO = 24 and BW = 4T, the CPU performs better than PIM at OC = **612** or higher. This marks the crossover point and sets the boundaries of a favorable region for PIM for this configuration. Note the placement of the OR, AND and MPY operations shown in Figure 4.2 along the x-axis. Clearly, OR (OC = 32) and ADD (OC = 144) are to the left of the crossover point and MPY (OC = 3104) to the right. The left region is where PIM is the favorable choice, while the right region is where CPU wins.

The crossover point shifts to the right for different DIO values. For instance, for MAT = 1024 and BW = 1024, the crossover point shifts roughly from OC = 2500 to OC = 5000 for DIO = 24 to DIO = 48, respectively. Thus, it is the algorithmic interplay of OC and DIO (along with other technological and architectural factors) that determines the throughput of PIM relative to that of traditional CPU/GPU computing.

4.1.3.2 Comparison of Power-limited Throughput

The maximum throughput for PIM or the CPU is limited by the thermal design power (TDP). For PIM, the throughput depends in turn on the energy per unit of computation, which is the energy spent during a single computation cycle (E^{PIM}) for OC = 1. We quantify the Power-Limited (PL) Throughput as:

Table 4.3. Power-related model parameters of the Bitlet model.

Parameter name	Notation	Value(s)	Type
Energy for PIM op (OC=1)	E^{PIM}	0.1pJ [68]	Technological
Energy for memory bit transfer	E^{CPU}	15pJ [85]	Technological
Thermal design power	TDP	20-160W	Architectural

$$PL\text{-}Throughput\text{-}PIM(Op) \; = \; Min(\frac{ROW \times MAT}{OC \times CT}, \frac{TDP}{E^{PIM} \times OC}). \tag{4.3}$$

On the CPU front, the energy per bit transfer (E^{CPU}) between the CPU and the memory determines the efficiency of CPU computations. We assume that the CPU compute energy is significantly lower than the data transfer energy. This aligns with our focus on identifying the strengths of PIM rather than those of the CPU. The PL-Throughput for CPU computation can be expressed as:

$$PL\text{-}Throughput\text{-}CPU(Op) \; = \; Min(\frac{BW}{DIO}, \frac{TDP}{E^{CPU} \times DIO}). \tag{4.4}$$

Table 4.3 summarizes the power-related model parameters, including their typical values or range of values. Figure 4.3 shows the maximum throughput possible for different configurations of PIM and CPU, under a given power budget.

For PIM, a maximum of **1950** $MATs$ can be accommodated at the power envelope of 20W. Increasing the number of MATs does not increase the throughput any further since the power budget of the system is the main limiter. PIM will support higher throughput only if we increase the power budget. For example, at 40W up to 3900 memory arrays (MATs) can be active at any given time.

For the CPU, the energy cost of data transfer limits the PL-Throughput-CPU. We assume here a BW = 16T. With the power limitation of 20W, the CPU can deliver 55 GOPS at DIO = 24. At the power budget of 40W, 111 GOPS are possible, and 444 GOPS at 160W. Compare this against the raw (with no power limitation) CPU throughput, which is 682 GOPS at DIO of 24.

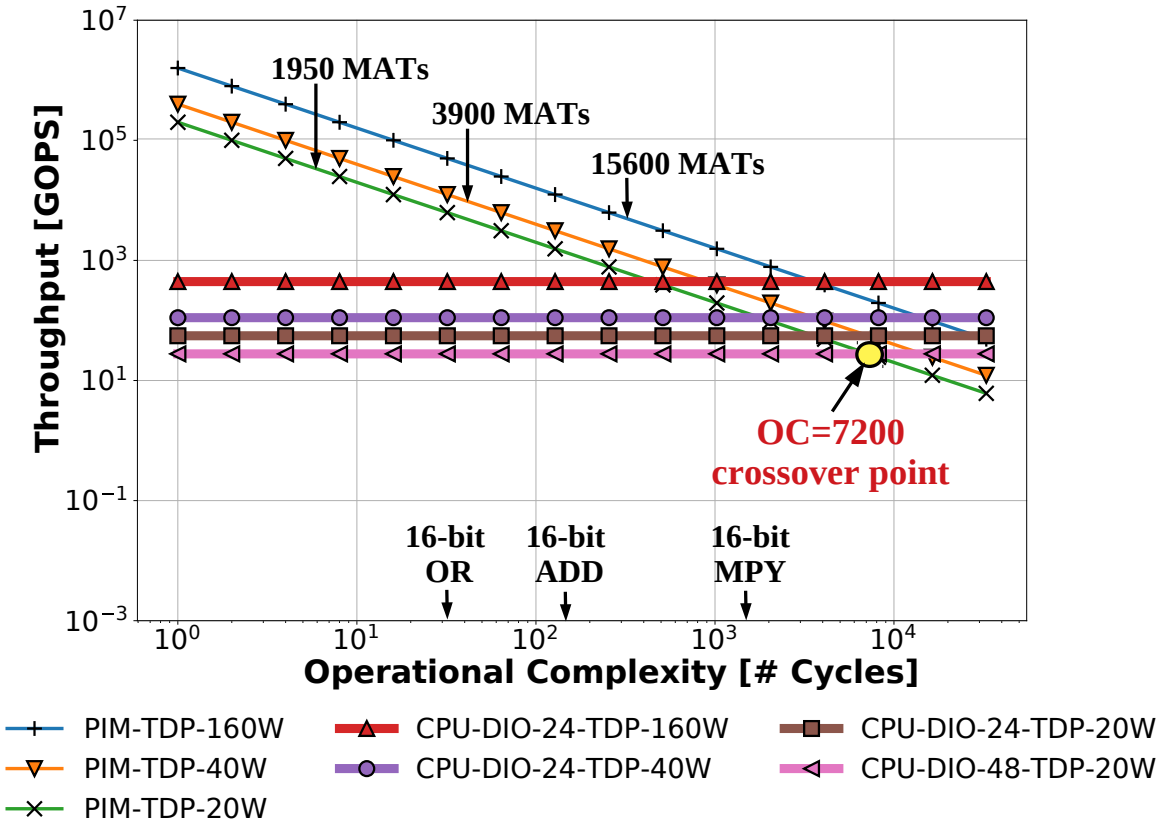


Figure 4.3. Throughput comparison of CPU vs. PIM under power limits. For the case of PIM, the figure shows the number of MATs, i.e. the memory arrays, permissible under a power limit.

The values of the energy parameters E^{PIM} and E^{CPU} affect the relative energy efficiency of PIM versus CPU. For example, consider the case of a single-bit NOR operation, where OC = 1 (a single MAGIC operation) and DIO = 3 (2 input and 1 output bits). In this case, PIM consumes 1xE^{PIM} = 0.1pJ while the CPU consumes 3xE^{CPU} = 45pJ. Overall, for this example, the CPU energy consumption is approximately **450X** higher than that of PIM.

However, as OC increases, the relative efficiency of PIM decreases. Now consider a 16-bit addition operation where OC = 144 and DIO = 48. For this operation, PIM consumes 14.4pJ (= 144x0.1) and the CPU consumes 720pJ (= 48x15). Thus, for a 16-bit ADD, PIM is **50X** (= 720/14.4) more energy efficient than the CPU. For the limiting case of OC = 7200 or higher (720pJ/0.1pJ = 7200), PIM becomes less attractive than CPU with respect to energy efficiency. However note that the specific values of the energy parameters affect the relative

merits of employing either PIM or a CPU.

4.2 Current Limitations of Bitlet Model

The Bitlet model includes several parameters for exploration of emerging systems that will employ PIM. The most important are operational complexity and data in-out. These along with the other supporting parameters allow various trade-off and limitation studies on PIM. We leave a more thorough exploration of all the outlined model parameters (with their possible values) for future work.

Architecture and technology will likely undergo changes. For instance, emerging memory interfaces such as high bandwidth memories (HBM) [85] will provide higher bandwidths. Similarly, the cycle time for new memories is constantly improving (sub 10 ns levels). The model analysis still applies to different bandwidth and memory cycle time values, but the trade-offs and break-even point will vary.

We also expect innovations that reduce the complexity of PIM-based operations. Such innovations include n-input NOR gates [110] and new logic gates [35]. The Bitlet model is useful to understand the extent of overall PIM throughput improvements that are anticipated on these fronts.

CPU arithmetic complexity and data reuse can be taken into account by either integrating or using the Bitlet model with existing models such as the Roofline model [117]. This will be useful for laying out the role of PIM in heterogeneous computing setups. We leave this exploration for future work.

Finally, although we compare PIM only against the CPU, the model assumptions and the comparisons can be extended to GPUs.

4.3 Related Work

Since the finding of the memristor [108]researchers have explored the use of new memories for computations.

4.3.1 In-Memory Computational Units

Initial efforts [71, 101, 33] described computing using logic operations on memory cells. Here the inputs were the applied voltages across the memory cells, and the outputs were stored inside the memory cells. Others [70]] explored using the stored data within the array as inputs and the output is sent as voltages. Later efforts [13, 99, 69, 64] investigated 'stateful logic' wherein the inputs and the outputs are all stored in the form of resistance values of the memory cells of the array.

Although there has been a steady reduction in over- heads, different efforts still differ in the peripheral circuitry, the voltages applied, how they use the rows and columns of the memory arrays, etc. These implementation intricacies make it hard to analyze, in a general manner, the capabilities and weaknesses of in-memory computing.

While generality of computations has steadily been achieved, it is being increasingly realized that there are limits on the complexity of the operations that can be per- formed effectively inside the memories. Simple operations (like OR, AND, NOR) are much easier and faster than complex operations like multiplication.

Figure 4.1 shows how bit lengths together with the complexity of operations can affect the number of operations of in-memory computing. For instance, multiplication has an kn^2 complexity and hence the operations grow much more rapidly than addition whose complexity is linear with n.

While complexity is one factor, it is widely acknowl- edged that the ability to process data inside the arrays is unique and holds the potential to reduce the amount of data to be moved. Reducing data movement saves external memory bandwidth and energy.

In this work, we show how these factors (operation complexity and data reduction) have a first-order impact on the prospects of using in-memory computing.

4.3.2 Analytical Computing Models

Historically, higher level computing models have been successful in promoting and developing an understanding of different forms of computing [39, 42, 117, 27]. They enable analyzing the performance, power, area trade-offs and complement full- fledged benchmarking.

For instance, the GPU computing model [42] gave a con- cise description of how the highly multi-threaded execution engines work. Similarly, heterogeneous-ISA multi-core CPU model [39] provided a clear view of the capabilities and limits of a system with asymmetric cores. Recent work on dark-silicon provided estimates of the fraction of a chip that can be active given the power limitations [27].

However, these past models or any known abstractions cannot be simply reapplied for in-memory computing for reasons discussed below.

Existing models for CPU/GPU [117] have primarily fo- cused on data reuse and arithmetic intensity. Reuse is maxi- mized, since otherwise, the data needs to be brought all the way from the memory. Arithmetic intensity (flops per bytes) is maximized in turn to benefit more from data reuse and to improve performance. The models also capture the system objective to reduce memory bandwidth demand.

In stark contrast, the in-memory computing does not heavily rely on data locality for performance. In fact, it relies more on internal bandwidth for computing rather than external. Furthermore, in a heterogeneous setting, in- memory computing can reduce the external memory band- width demand from an accompanying CPU/GPU.

Since, both the optimizing objectives and the nature of computational units involved (as discussed earlier), are different in case of in-memory computing, it warrants a rethink and a new model.

4.4 Conclusions

The paper motivates and describes an analytical model for PIM called Bitlet. We show how to use the model to find the beneficial use cases for PIM and understand the related trade-offs and limits, in a parameterized fashion. Using the model, we found precise crossover points with specific operational complexity. For example, for *OC* under 612, PIM using 1,000 memory arrays has advantages over traditional computing using 1Tbps bandwidth and *DIO* of 24. Similarly, we found crossover points from an energy efficiency perspective (*OC* = 7200 or lower for any operation with *DIO* of 48). We hope that the Bitlet model will provide a clearer view of the avenues for PIM in future systems.

4.5 Acknowledgement

Chapter 4, in part is currently being reviewed for publication in the *IEEE Computer Architecture Letters (CAL '19)*; Bitlet Model: Understanding the Sweet Spots and Limits of Processing in Memory; Kunal Korgaonkar, Ronny Ronen, Anupam Chattopadhyay and Shahar Kvatinsky, 2019. The dissertation author was the primary investigator and author of this material.

Bibliography

[1] N. Agarwal, L. S. Peh, and N. K. Jha. “In-Network Coherence Filtering: Snoopy coherence without broadcasts”. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2009, pp. 232–243. DOI: 10.1145/1669112.1669143.

[2] N. Agarwal, L. S. Peh, and N. K. Jha. “In-Network Snoop Ordering (INSO): Snoopy coherence on unordered interconnects”. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. Feb. 2009, pp. 67–78. DOI: 10.1109/HPCA.2009.4798238.

[3] J. Ahn, S. Yoo, and K. Choi. “DASCA: Dead Write Prediction Assisted STT-RAM Cache Architecture”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 25–36. DOI: 10.1109/HPCA.2014.6835944.

[4] “AMD. Phenom II processor model”. In: (). URL: http://www.amd.com/en-us/products/processors/desktop/phenom-ii.

[5] ARM. “Armv8-A architecture evolution, persistent memory instructions”. In: *ARM blog* (). URL: https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-evolution.

[6] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. “Bztree: A High-performance Latch-free Range Index for Non-volatile Memory”. In: *Proc. VLDB*

Endow. 11.5 (Jan. 2018), pp. 553–565. ISSN: 2150-8097. DOI: 10.1145/3164135.3164147. URL: https://doi.org/10.1145/3164135.3164147.

[7] Hillel Avni and Trevor Brown. "Persistent Hybrid Transactional Memory for Databases". In: *Proc. VLDB Endow.* 10.4 (2016), pp. 409–420. ISSN: 2150-8097. DOI: 10.14778/3025111.3025122. URL: https://doi.org/10.14778/3025111.3025122.

[8] SPEC CPU 2006 Benchmarks. In: (). URL: http://www.spec.org/cpu2006.

[9] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay. "ReVAMP: ReRAM based VLIW architecture for in-memory computing". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. Mar. 2017, pp. 782–787. DOI: 10.23919/DATE.2017.7927095.

[10] E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin, Mark D. Hill, and David A. Wood. "Multicast Snooping: A New Coherence Method Using a Multicast Address Network". In: *Proceedings of the 26th Annual International Symposium on Computer Architecture*. ISCA '99. Atlanta, Georgia, USA: IEEE Computer Society, 1999, pp. 294–304. ISBN: 0-7695-0170-2. DOI: 10.1145/300979.301004. URL: http://dx.doi.org/10.1145/300979.301004.

[11] R. Bisisani, Andreas Nowatzyk, and M. Ravishankar. "Coherent Shared Memory on a Message Passing Machine". In: *Proceedings of the 1989 International Conference on Parallel Processing*. Jan. 1989.

[12] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. "The Parallel Persistent Memory Model". In: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. SPAA '18. Vienna, Austria: ACM, 2018, pp. 247–258. ISBN: 978-1-4503-5799-9. DOI: 10.1145/3210377.3210381. URL: http://doi.acm.org/10.1145/3210377.3210381.

[13] Julien Borghetti, Gregory S Snider, Philip J Kuekes, Jianhua Joshua Yang, Duncan Stewart, and Stan Williams. “Memristive switches enable stateful logic operations via material implication”. In: *Nature* 464 (Apr. 2010), pp. 873–6. DOI: 10.1038/nature08940.

[14] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. “Atlas: Leveraging Locks for Non-volatile Memory Consistency”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. Portland, Oregon, USA: ACM, 2014, pp. 433–452. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660224. URL: http://doi.acm.org/10.1145/2660193.2660224.

[15] Mu-Tien Chang, Paul Rosenfeld, Shih-Lien Lu, and Bruce Jacob. “Technology Comparison for Large Last-level Caches (L3Cs): Low-leakage SRAM, Low Write-energy STT-RAM, and Refresh-optimized eDRAM”. In: *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. HPCA ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 143–154. ISBN: 978-1-4673-5585-8. DOI: 10.1109/HPCA.2013.6522314. URL: http://dx.doi.org/10.1109/HPCA.2013.6522314.

[16] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. “Introducing Hierarchy-awareness in replacement and bypass algorithms for last-level caches”. In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2012, pp. 293–304.

[17] H. Cheng, J. Zhao, J. Sampson, M. J. Irwin, A. Jaleel, Y. Lu, and Y. Xie. “LAP: Loop-Block Aware Inclusion Properties for Energy-Efficient Asymmetric Last Level Caches”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 103–114. DOI: 10.1109/ISCA.2016.19.

[18] A. Chintaluri, H. Naeimi, S. Natarajan, and A. Raychowdhury. “Analysis of Defects and Variations in Embedded Spin Transfer Torque (STT) MRAM Arrays”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 6.3 (Sept. 2016), pp. 319–329. ISSN: 2156-3357. DOI: 10.1109/JETCAS.2016.2547779.

[19] K. C. Chun, H. Zhao, J. D. Harms, T. Kim, J. Wang, and C. H. Kim. “A Scaling Roadmap and Performance Evaluation of In-Plane and Perpendicular MTJ Based STT-MRAMs for High-Density Cache Memory”. In: *IEEE Journal of Solid-State Circuits* 48.2 (Feb. 2013), pp. 598–610. ISSN: 0018-9200. DOI: 10.1109/JSSC.2012.2224256.

[20] 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: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’11. Newport Beach, California, USA: ACM, 2011, pp. 105–118. ISBN: 978-1-4503-0266-1. DOI: http://doi.acm.org/10.1145/1950365.1950380. URL: http://doi.acm.org/10.1145/1950365.1950380.

[21] Nachshon Cohen, David T. Aksun, and James R. Larus. “Object-oriented Recovery for Non-volatile Memory”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 153:1–153:22. ISSN: 2475-1421. DOI: 10.1145/3276523. URL: http://doi.acm.org/10.1145/3276523.

[22] 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. Big Sky, Montana, USA: ACM, 2009, pp. 133–146. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629589. URL: http://doi.acm.org/10.1145/1629575.1629589.

[23] I. Cutress. “The Intel Skylake Mobile and Desktop Launch, with Architecture Analysis. September 2015.” In: (). URL: http://www.anandtech.com/show/9582/intel-skylakemobile-desktop-launch-architecture-analysis/5..

[24] B. K. Daya, C. H. O. Chen, S. Subramanian, W. C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L. S. Peh. “SCORPIO: A 36-core research chip demonstrating snoopy coherence on a scalable mesh NoC with in-network ordering”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. June 2014, pp. 25–36. DOI: 10.1109/ISCA.2014.6853232.

[25] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. “Data Tiering in Heterogeneous Memory Systems”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys ’16. London, United Kingdom: ACM, 2016, 15:1–15:16. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901344. URL: http://doi.acm.org/10.1145/2901318.2901344.

[26] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. “Neural Cache: Bit-serial In-cache Acceleration of Deep Neural Networks”. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA ’18. Los Angeles, California: IEEE Press, 2018, pp. 383–396. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00040. URL: https://doi.org/10.1109/ISCA.2018.00040.

[27] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. “Dark Silicon and the End of Multicore Scaling”. In: *IEEE Micro* 32.3 (May 2012), pp. 122–134. ISSN: 0272-1732. DOI: 10.1109/MM.2012.17.

[28] Colin J. Fidge. “Timestamps in Message-Passing Systems that Preserve the Partial Ordering”. In: *11th Australian Computer Science Conference*. 1988, pp. 55–66.

[29] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. “A Persistent Lock-free Queue for Non-volatile Memory”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18. Vienna, Austria: ACM, 2018, pp. 28–40. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178490. URL: http://doi.acm.org/10.1145/3178487.3178490.

[30] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. “In-Memory Data Parallel Processor”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’18. Williamsburg, VA, USA: ACM, 2018, pp. 1–14. ISBN: 978-1-4503-4911-6. DOI: 10.1145/3173162.3173171. URL: http://doi.acm.org/10.1145/3173162.3173171.

[31] R. M. Fujimoto. “The Virtual Time Machine”. In: *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’89. Santa Fe, New Mexico, USA: ACM, 1989, pp. 199–208. ISBN: 0-89791-323-X. DOI: 10.1145/72935.72957. URL: http://doi.acm.org/10.1145/72935.72957.

[32] S. Fujita, H. Noguchi, K. Ikegami, S. Takeda, K. Nomura, and K. Abe. “Novel memory hierarchy with e-STT-MRAM for near-future applications”. In: *2017 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. Apr. 2017, pp. 1–2. DOI: 10.1109/VLSI-DAT.2017.7939700.

[33] P. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli. “The Programmable Logic-in-Memory (PLiM) computer”. In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 427–432.

[34] J. Gaur, M. Chaudhuri, and S. Subramoney. “Bypass and insertion algorithms for exclusive last-level caches”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. June 2011, pp. 81–92.

[35] Saransh Gupta, Mohsen Imani, and Tajana Rosing. “FELIX: Fast and Energy-efficient Logic in Memory”. In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD ’18. San Diego, California: ACM, 2018, 55:1–55:7. ISBN: 978-1-4503-5950-4. DOI: 10.1145/3240765.3240811. URL: http://doi.acm.org/10.1145/3240765.3240811.

[36] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5 (1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: http://doi.acm.org/10.1145/42411.42415.

[37] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky. “IMAGING: In-Memory AlGorithms for Image processiNG”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.12 (Dec. 2018), pp. 4258–4271. ISSN: 1549-8328. DOI: 10.1109/TCSI.2018.2846699.

[38] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky. “Not in Name Alone: A Memristive Memory Processing Unit for Real In-Memory Processing”. In: *IEEE Micro* 38.5 (Sept. 2018), pp. 13–21. ISSN: 0272-1732. DOI: 10.1109/MM.2018.053631137.

[39] M. D. Hill and M. R. Marty. “Amdahl’s Law in the Multicore Era”. In: *Computer* 41.7 (July 2008), pp. 33–38. ISSN: 0018-9162. DOI: 10.1109/MC.2008.209.

[40] Mark Hill and Vijay Janapa Reddi. “Gables: A Roofline Model for Mobile SoCs”. In: Feb. 2019, pp. 317–330. DOI: 10.1109/HPCA.2019.00047.

[41] J. Holliday, D. Agrawal, and A. El Abbadi. “Partial database replication using epidemic communication”. In: *Proceedings 22nd International Conference on Distributed Computing Systems*. July 2002, pp. 485–493. DOI: 10.1109/ICDCS.2002.1022298.

[42] Sunpyo Hong and Hyesoon Kim. “An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness”. In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA ’09. Austin, TX, USA: ACM, 2009, pp. 152–163. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555775. URL: http://doi.acm.org/10.1145/1555754.1555775.

[43] “HPCG Benchmark.” In: (). URL: http://hpcg-benchmark.org/.

[44] R. B. Hur and S. Kvatinsky. “Memory Processing Unit for in-memory processing”. In: *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. July 2016, pp. 171–172. DOI: 10.1145/2950067.2950086.

[45] H. Iiboshi and T. Ugawa. “Towards Model Checking Library for Persistent Data Structures”. In: *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. Aug. 2018, pp. 119–120. DOI: 10.1109/NVMSA.2018.00032.

[46] Mohsen Imani, Saransh Gupta, and Tajana Rosing. “Ultra-Efficient Processing In-Memory for Data Intensive Applications”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC ’17. Austin, TX, USA: ACM, 2017, 6:1–6:6. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062337. URL: http://doi.acm.org/10.1145/3061639.3062337.

[47] Micron Technology Inc. “Calculating Memory System Power for DDR3. Micron Technical Note TN-41-01.” In: (). URL: http://www.micron.com/products/support/power-calc.

[48] Intel and Micron. “3D XPoint Technology”. In: (2017). URL: http://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[49] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. “Failure-Atomic Persistent Memory Updates via JUSTDO Logging”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: ACM, 2016, pp. 427–442. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872410. URL: http://doi.acm.org/10.1145/2872362.2872410.

[50] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. “Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model”. In: *Proceedings of the 30th International Conference on Distributed Computing*. Ed. by Cyril Gavoille and David Ilcinkas. DISC ’16. Paris, France: Springer Berlin Heidelberg, 2016, pp. 313–327. ISBN: 978-3-662-53426-7. DOI: 10.1007/978-3-662-53426-7_23. URL: http://dx.doi.org/10.1007/978-3-662-53426-7_23.

[51] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. “Adaptive insertion policies for managing shared caches”. In: *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Oct. 2008, pp. 208–219.

[52] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. “High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)”. In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA ’10. Saint-Malo, France: ACM, 2010, pp. 60–71. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815971. URL: http://doi.acm.org/10.1145/1815961.1815971.

[53] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. “Efficient Persist Barriers for Multicores”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 660–671. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830805. URL: http://doi.acm.org/10.1145/2830772.2830805.

[54] D. Kang, S. Baek, J. Choi, D. Lee, S. H. Noh, and O. Mutlu. “Amnesic cache management for non-volatile memory”. In: *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. May 2015, pp. 1–13. DOI: 10.1109/MSST.2015.7208291.

[55] A. Kawahara, K. Kawai, Y. Ikeda, Y. Katoh, R. Azuma, Y. Yoshimoto, K. Tanabe, Z. Wei, T. Ninomiya, K. Katayama, R. Yasuhara, S. Muraoka, A. Himeno, N. Yoshikawa, H. Murase, K. Shimakawa, T. Takagi, T. Mikawa, and K. Aono. “Filament scaling forming technique and level-verify-write scheme with endurance over 107 cycles in ReRAM”. In: *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. Feb. 2013, pp. 220–221. DOI: 10.1109/ISSCC.2013.6487708.

[56] S. M. Khan, Y. Tian, and D. A. Jimenez. “Sampling Dead Block Prediction for Last-Level Caches”. In: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2010, pp. 175–186. DOI: 10.1109/MICRO.2010.24.

[57] A V Khvalkovskiy, D Apalkov, S Watts, R Chepulskii, R S Beach, A Ong, X Tang, A Driskill-Smith, W H Butler, P B Visscher, D Lottis, E Chen, V Nikitin, and M Krounbi. “Basic principles of STT-MRAM cell operation in memory arrays”. In: *Journal of Physics D: Applied Physics* 46.7 (Feb. 2013), p. 074001. DOI: 10.1088/0022-3727/46/7/074001. URL: https://doi.org/10.1088%2F0022-3727%2F46%2F7%2F074001.

[58] J. Kim, B. Tuohy, C. Ma, W. H. Choi, I. Ahmed, D. Lilja, and C. H. Kim. “Spin-Hall effect MRAM based cache memory: A feasibility study”. In: *2015 73rd Annual Device Research Conference (DRC)*. June 2015, pp. 117–118. DOI: 10.1109/DRC.2015.7175583.

[59] E. Kitagawa, S. Fujita, K. Nomura, H. Noguchi, K. Abe, K. Ikegami, T. Daibou, Y. Kato, C. Kamata, S. Kashiwada, N. Shimomura, J. Ito, and H. Yoda. “Impact of ultra low power and fast write operation of advanced perpendicular MTJ on power reduction for

high-performance mobile CPU". In: *2012 International Electron Devices Meeting*. Dec. 2012, pp. 29.4.1–29.4.4. DOI: 10.1109/IEDM.2012.6479129.

[60] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. "Delegated persist ordering". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783761.

[61] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. "Language-level Persistency". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 481–493. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080229. URL: http://doi.acm.org/10.1145/3079856.3080229.

[62] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. "High-Performance Transactions for Persistent Memories". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 399–411. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872381. URL: http://doi.acm.org/10.1145/2872362.2872381.

[63] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. "Evaluating STT-RAM as an energy-efficient main memory alternative". In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 256–267. DOI: 10.1109/ISPASS.2013.6557176.

[64] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. "MAGIC—Memristor-Aided Logic". In: *IEEE Transactions on Circuits*

and Systems II: Express Briefs 61.11 (Nov. 2014), pp. 895–899. ISSN: 1549-7747. DOI: 10.1109/TCSII.2014.2357292.

[65] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.10 (Oct. 2014), pp. 2054–2066. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2013.2282132.

[66] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[67] Anders Landin, Erik Hagersten, and Seif Haridi. "Race-free Interconnection Networks and Multiprocessor Consistency". In: *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ISCA '91. Toronto, Ontario, Canada: ACM, 1991, pp. 106–115. ISBN: 0-89791-394-9. DOI: 10.1145/115952.115964. URL: http://doi.acm.org/10.1145/115952.115964.

[68] Mario Lanza, H.-S. Philip Wong, Eric Pop, Daniele Ielmini, Dimitri Strukov, Brian C. Regan, Luca Larcher, Marco A. Villena, J. Joshua Yang, Ludovic Goux, Attilio Belmonte, Yuchao Yang, Francesco M. Puglisi, Jinfeng Kang, Blanka Magyari-Köpe, Eilam Yalon, Anthony Kenyon, Mark Buckwell, Adnan Mehonic, Alexander Shluger, Haitong Li, Tuo-Hung Hou, Boris Hudec, Deji Akinwande, Ruijing Ge, Stefano Ambrogio, Juan B. Roldan, Enrique Miranda, Jordi Suñe, Kin Leong Pey, Xing Wu, Nagarajan Raghavan, Ernest Wu, Wei D. Lu, Gabriele Navarro, Weidong Zhang, Huaqiang Wu, Runwei Li, Alexander Holleitner, Ursula Wurstbauer, Max C. Lemme, Ming Liu, Shibing Long, Qi Liu, Hangbing Lv, Andrea Padovani, Paolo Pavan, Ilia Valov, Xu Jing, Tingting Han, Kaichen Zhu, Shaochuan Chen, Fei Hui, and Yuanyuan Shi. "Recommended Methods to Study Resistive Switching Devices". In: *Advanced Electronic Materials* 5.1 (2019),

p. 1800143. DOI: 10.1002/aelm.201800143. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/aelm.201800143. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/aelm.201800143.

[69] E. Lehtonen, J. H. Poikonen, and M. Laiho. "Two memristors suffice to compute all boolean functions". In: *Electronics Letters* 46.3 (Feb. 2010), pp. 239–240. ISSN: 0013-5194. DOI: 10.1049/el.2010.3407.

[70] Yifat Levy, Jehoshua Bruck, Yuval Cassuto, Eby G. Friedman, Avinoam Kolodny, Eitan Yaakobi, and Shahar Kvatinsky. "Logic Operations in Memory Using a Memristive Akers Array". In: *Microelectron. J.* 45.11 (2014), pp. 1429–1437. ISSN: 0026-2692. DOI: 10.1016/j.mejo.2014.06.006. URL: http://dx.doi.org/10.1016/j.mejo.2014.06.006.

[71] Eike Linn, R Rosezin, Stefan Tappertzhofen, U Böttger, and Rainer Waser. "Beyond von Neumann - Logic operations in passive crossbar arrays alongside memory operations". In: *Nanotechnology* 23 (July 2012), p. 305205. DOI: 10.1088/0957-4484/23/30/305205.

[72] M. Lis, K. S. Shim, M. H. Cho, and S. Devadas. "Memory coherence in the age of multicores". In: *2011 IEEE 29th International Conference on Computer Design (ICCD)*. Oct. 2011, pp. 1–8. DOI: 10.1109/ICCD.2011.6081367.

[73] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. "iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2018, pp. 258–270. DOI: 10.1109/MICRO.2018.00029.

[74] 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 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA:

ACM, 2014, pp. 455–470. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541957. URL: http://doi.acm.org/10.1145/2541940.2541957.

[75] Sasikanth Manipatruni, Dmitri E. Nikonov, and Ian A. Young. "Energy-delay performance of giant spin Hall effect switching for dense magnetic memory". In: *Applied Physics Express* 7.10 (Sept. 2014), p. 103001. DOI: 10.7567/apex.7.103001. URL: https://doi.org/10.7567%2Fapex.7.103001.

[76] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. "Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory". In: *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, 2017. URL: https://www.usenix.org/conference/hotstorage17/program/presentation/marathe.

[77] M. M. K. Martin, M. D. Hill, and D. A. Wood. "Token Coherence: decoupling performance and correctness". In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* June 2003, pp. 182–193. DOI: 10.1109/ISCA.2003.1206999.

[78] Milo M. K. Martin, Daniel J. Sorin, Anatassia Ailamaki, Alaa R. Alameldeen, Ross M. Dickson, Carl J. Mauer, Kevin E. Moore, Manoj Plakal, Mark D. Hill, and David A. Wood. "Timestamp Snooping: An Approach for Extending SMPs". In: *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IX. Cambridge, Massachusetts, USA: ACM, 2000, pp. 25–36. ISBN: 1-58113-317-0. DOI: 10.1145/378993.378998. URL: http://doi.acm.org/10.1145/378993.378998.

[79] Friedemann Mattern. "Virtual Time and Global States of Distributed Systems". In: *PARALLEL AND DISTRIBUTED ALGORITHMS*. North-Holland, 1988, pp. 215–226.

[80] 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. Xi’an, China: ACM, 2017, pp. 135–148. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037730. URL: http://doi.acm.org/10.1145/3037697.3037730.

[81] Rajeev Balasubramonian Naveen Muralimanohar and Norman P. Jouppi. “CACTI 6.0: A Tool to Model Large Caches, HP Labs Technical Report HPL-2009-85, HP Laboratories, 2009”. In: ().

[82] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. “Dalí: A Periodically Persistent Hash Map”. In: *31st International Symposium on Distributed Computing (DISC 2017)*. Ed. by Andréa W. Richa. Vol. 91. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 37:1–37:16. ISBN: 978-3-95977-053-8. DOI: 10.4230/LIPIcs.DISC.2017.37. URL: http://drops.dagstuhl.de/opus/volltexte/2017/8014.

[83] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita. “7.5 A 3.3ns-access-time 71.2uW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture”. In: *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*. Feb. 2015, pp. 1–3. DOI: 10.1109/ISSCC.2015.7062963.

[84] H. Noguchi, K. Ikegami, S. Takaya, E. Arima, K. Kushida, A. Kawasumi, H. Hara, K. Abe, N. Shimomura, J. Ito, S. Fujita, T. Nakada, and H. Nakamura. “7.2 4Mb STT-MRAM-based cache with memory-access-aware power optimization and write-verify-

write / read-modify-write scheme". In: *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. Jan. 2016, pp. 132–133. DOI: 10.1109/ISSCC.2016.7417942.

[85] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. "Fine-grained DRAM: Energy-efficient DRAM for Extreme Bandwidth Systems". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Massachusetts: ACM, 2017, pp. 41–54. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3124545. URL: http://doi.acm.org/10.1145/3123939.3124545.

[86] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. "MARSS: A Full System Simulator for Multicore x86 CPUs". In: *Proceedings of the 48th Design Automation Conference*. DAC '11. San Diego, California: ACM, 2011, pp. 1050–1055. ISBN: 978-1-4503-0636-2. DOI: 10.1145/2024724.2024954. URL: http://doi.acm.org/10.1145/2024724.2024954.

[87] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. "Memory Persistency". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 265–276. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665712.

[88] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. "Memory Persistency". In: *Proceeding of the 41st Annual International Symposium on Computer Architecture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 265–276. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665712.

[89] Azalea Raad and Viktor Vafeiadis. "Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model". In: *Proc. ACM Program. Lang.*

2.OOPSLA (2018), 137:1–137:27. ISSN: 2475-1421. DOI: 10.1145/3276507. URL: http://doi.acm.org/10.1145/3276507.

[90] P. Ranganathan. “From Microprocessors to Nanostores: Rethinking Data-Centric Systems”. In: *Computer* 44.1 (Jan. 2011), pp. 39–48. ISSN: 0018-9162. DOI: 10.1109/MC.2011.18.

[91] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. “Phase-change Random Access Memory: A Scalable Technology”. In: *IBM J. Res. Dev.* 52.4 (2008), pp. 465–479. ISSN: 0018-8646. DOI: 10.1147/rd.524.0465. URL: http://dx.doi.org/10.1147/rd.524.0465.

[92] P. F. Reynolds, C. Williams, and R. R. Wagner. “Isotach networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 8.4 (Apr. 1997), pp. 337–348. ISSN: 1045-9219. DOI: 10.1109/71.588601.

[93] Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. “From Causal to Z-linearizable Transactional Memory”. In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’07. Portland, Oregon, USA: ACM, 2007, pp. 340–341. ISBN: 978-1-59593-616-5. DOI: 10.1145/1281100.1281162. URL: http://doi.acm.org/10.1145/1281100.1281162.

[94] Andy Rudolf. “Processor support for NVM programming”. In: *NVM summit* (). URL: http://www.snia.org/sites/default/files/AndyRudoff_Processor_Support_NVM.pdf.

[95] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. “Willow: A User-programmable SSD”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and*

Implementation. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 67–80. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685055.

[96] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology". In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2017, pp. 273–287.

[97] Mustafa Shihab, Jie Zhang, Shuwen Gao, Josep Sloan, and Myoungsoo Jung. "Couture: Tailoring STT-MRAM for Persistent Main Memory". In: *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*. Savannah, GA: USENIX Association, 2016. URL: https://www.usenix.org/conference/inflow16/workshop-program/presentation/couture-tailoring-stt-mram-persistent-main-memory.

[98] Keun Sup Shim, Myong Hyon Cho, Mieszko Lis, Omer Khan, and Srinivas Devadas. "Library Cache Coherence". In: 2011. URL: http://hdl.handle.net/1721.1/62580.

[99] S. Shin, K. Kim, and S. Kang. "Reconfigurable Stateful nor Gate for Large-Scale Logic-Array Integrations". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 58.7 (July 2011), pp. 442–446. ISSN: 1549-7747. DOI: 10.1109/TCSII.2011.2158253.

[100] Seunghee Shin, James Tuck, and Yan Solihin. "Hiding the Long Latency of Persist Barriers Using Speculative Execution". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 175–186. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080240. URL: http://doi.acm.org/10.1145/3079856.3080240.

[101] A. Siemon, S. Menzel, R. Waser, and E. Linn. "A Complementary Resistive Switch-Based Crossbar Array Adder". In: *IEEE Journal on Emerging and Selected Topics in*

Circuits and Systems 5.1 (Mar. 2015), pp. 64–74. ISSN: 2156-3357. DOI: 10.1109/JETCAS.2015.2398217.

[102] J. Sim, J. Lee, M. K. Qureshi, and H. Kim. "FLEXclusion: Balancing cache capacity and on-chip bandwidth via Flexible Exclusion". In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. June 2012, pp. 321–332. DOI: 10.1109/ISCA.2012.6237028.

[103] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. "Cache Coherence for GPU Architectures". In: *IEEE Micro* 34.3 (May 2014), pp. 69–79. ISSN: 0272-1732. DOI: 10.1109/MM.2014.4.

[104] M. Singhal and M. Raynal. "Logical Time: Capturing Causality in Distributed Systems". In: *Computer* 29 (Feb. 1996), pp. 49–56. ISSN: 0018-9162. DOI: 10.1109/2.485846. URL: doi.ieeecomputersociety.org/10.1109/2.485846.

[105] Mukesh Singhal and Ajay Kshemkalyani. "An efficient implementation of vector clocks". In: *Information Processing Letters* 43.1 (1992), pp. 47–52. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(92)90028-T. URL: http://www.sciencedirect.com/science/article/pii/002001909290028T.

[106] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan. "Relaxing non-volatility for fast and energy-efficient STT-RAM caches". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. Feb. 2011, pp. 50–61. DOI: 10.1109/HPCA.2011.5749716.

[107] K. Strauss, X. Shen, and J. Torrellas. "Uncorq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. Dec. 2007, pp. 327–342. DOI: 10.1109/MICRO.2007.37.

[108] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. “The missing memristor found”. In: *nature* 453.7191 (2008), pp. 80–83.

[109] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen. “A novel architecture of the 3D stacked MRAM L2 cache for CMPs”. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. Feb. 2009, pp. 239–249. DOI: 10.1109/HPCA.2009.4798259.

[110] Valerio Tenace, Roberto Rizzo, Debjyoti Bhattacharjee, Anupam Chattopadhyay, and Andrea Calimera. “SAID: A Supergate-Aided Logic Synthesis Flow for Memristive Crossbars”. In: 2019.

[111] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. “Multi2Sim: A simulation framework for CPU-GPU computing”. In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Sept. 2012, pp. 335–344.

[112] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy Campbell. “Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory”. In: *Proceedings of the 9th USENIX Conference on File and Storage Technologies*. FAST ’11. San Jose, CA, USA: USENIX Association, Feb. 2011, pp. 5–5.

[113] Haris Volos, Andres Jaan Tack, and Michael M. Swift. “Mnemosyne: Lightweight Persistent Memory”. In: *ASPLOS ’11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. Newport Beach, California, USA: ACM, 2011.

[114] J. Wang, X. Dong, and Y. Xie. “OAP: An obstruction-aware cache management policy for STT-RAM last-level caches”. In: *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2013, pp. 847–852. DOI: 10.7873/DATE.2013.179.

[115] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie. "Adaptive placement and migration policy for an STT-RAM-based hybrid cache". In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2014, pp. 13–24. DOI: 10.1109/HPCA.2014.6835933.

[116] C. Williams, P. F. Reynolds, and B. R. de Supinski. "Delta coherence protocols". In: *IEEE Concurrency* 8.3 (July 2000), pp. 23–29. ISSN: 1092-3063. DOI: 10.1109/4434.865889.

[117] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[118] H. S. P. Wong, H. Y. Lee, S. Yu, Y. S. Chen, Y. Wu, P. S. Chen, B. Lee, F. T. Chen, and M. J. Tsai. "Metal Oxide RRAM". In: *Proceedings of the IEEE* 100.6 (June 2012), pp. 1951–1970. ISSN: 0018-9219. DOI: 10.1109/JPROC.2012.2190369.

[119] H.-S. P. Wong, C. Ahn, J. Cao, H.-Y. Chen, S. W. Fong, Z. Jiang, C. Neumann, S. Qin, J. Sohn, Y. Wu, S. Yu, X. Zheng, H. Li, J. A. Incorvia, S. B. Eryilmaz, and K. Okabe. "Stanford Memory Trends. Accessed November 8, 2016." In: (). URL: https://nano.stanford.edu/stanford-memory-trends.

[120] Jiang Wu. "Checkpointing and recovery in distributed and database systems". In: (2011).

[121] 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)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 323–338. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[122] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 478–496. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132761. URL: http://doi.acm.org/10.1145/3132747.3132761.

[123] J. Zhan, O. Kayıran, G. H. Loh, C. R. Das, and Y. Xie. "OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783731.

[124] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F. T. Chong. "Mellow Writes: Extending Lifetime in Resistive Memories through Selective Slow Write Backs". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 519–531. DOI: 10.1109/ISCA.2016.52.

[125] J. Zhao, O. Mutlu, and Y. Xie. "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Dec. 2014, pp. 153–165. DOI: 10.1109/MICRO.2014.47.

[126] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. "Energy reduction for STT-RAM using early write termination". In: *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*. Nov. 2009, pp. 264–268.