DENOVO: RETHINKING THE MEMORY HIERARCHY FOR DISCIPLINED PARALLELISM

BY

HYOJIN SUNG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Abstract

As multicore systems become widespread, both software and hardware face a major challenge in efficiently exploiting and implementing parallelism. While shared-memory remains a popular programming model due to its global address space, it is plagued with undisciplined programming practices that allow implicit communication and unstructured non-determinism. Such “wild” shared-memory behavior not only makes it difficult to test and maintain software but also complicates hardware, preventing it from scaling in a power-efficient manner. Recent research has proposed replacing the wild shared-memory programming models with a more disciplined approach.

The DeNovo project asks the following question: *if software is more disciplined, can we build more power-, performance-, and complexity-efficient shared-memory hardware?* Focusing on deterministic programs as a discipline to drive DeNovo, we first show that coherence and communication can be made much simpler and more efficient than the current state of the art. The resulting protocol is without transient states, invalidation traffic, directory sharer-lists, or false sharing - all significant sources of inefficiencies in existing protocols. Widening the software space further, we then show how DeNovo can support software with disciplined non-determinism without giving up its benefits for deterministic programs. The remaining challenge is to support synchronization accesses that are inherently “racy” on DeNovo without writer-initiated invalidation. We show that arbitrary synchronization can be supported on DeNovo with a simple yet efficient hardware mechanism, a big step toward our eventual goal of supporting legacy programs. Finally, we explore the potential for a comprehensive coherence solution that merges all previous DeNovo coherence mechanisms and adaptively switches between them depending on the level of “discipline” of software.

In summary, DeNovo shows the potential for commercially viable software-driven shared-memory systems with higher complexity-, performance-, and energy-efficiency than today’s software-oblivious hardware.

To Joonho and Irene

Acknowledgments

While this short acknowledgment section can never be enough to thank all of the people who supported and encouraged me to stay the course until the end of my Ph.D. journey, I will try my best to make do.

I would first like to express my deepest gratitude to my advisor, Sarita Adve, for her expert guidance and support through every step in my path, whether achievement or error. I have been extremely fortunate to learn from such an extraordinary researcher and inspiring teacher. Her undying enthusiasm for top-quality research, never-give-up attitude, and brilliant insight, as well as her rigorous attention to detail, are constant motivations for me. As a fellow female researcher, she was an exemplary mentor and role model, and I dearly cherish her words of wisdom. It was truly a great honor and privilege to have Sarita as my Ph.D. advisor.

I would also like to sincerely thank my Ph.D. committee, Marc Snir, Vikram Adve, Craig Zilles, Nicholas Carter, Sandhya Dwarkadas, and Daniel Sorin for their time and insightful feedback on my thesis. Their comprehensive and constructive comments greatly improved the coherence and completeness of my thesis. I offer special thanks to Arch Robison, my internship mentor at Intel, for giving me the opportunity to explore real-world parallel programming issues.

I thank Rakesh for our collaborations on the DeNovo project. Working with him was not only a great research and team experience, but also a lot of fun. I am also thankful to my other collaborators on the project, Byn, Nima, Rob Bocchino, and Rob Smolinski. Special thanks to my lab-mates Matt, Siva, Radha, Abdulrahman, Huzaifa, John, and Pradeep, for the vibrant research environment, interesting discussions, and of course, good times.

Many thanks to the friendly staff in our department, including Molly and Michelle, for helping me with all kinds of administrative tasks, from travel arrangements to conference room bookings, and Mary Beth, for being incredibly patient and supportive during my thesis deposit process.

My work has been supported by Intel and Microsoft through the Universal Parallel Computing Research Center (UPCRC) at Illinois, by Intel through the Illinois/Intel Parallelism Center at Illinois, by the National Science Foundation under grants CCF-1018796 and CCF-1302641, and by the Center for Future Architectures Research (C-FAR), which is one of six centers of STARnet, a semiconductor research corporation program sponsored by MARCO and DARPA. Extra special thanks to Mrs. Poppelbaum and Mr. and Mrs. Laursen for their generous donations through the Poppelbaum award and the Andrew and Shana Laursen Fellowship in the Department of Computer Science at Illinois, of which I was an honored recipient.

My warm thanks to the friends and acquaintances that I have made at the Korean Buddhist Student Association at Illinois. The meditations, prayers, and fun times that we shared helped me to keep a healthy balance in my early years in Champaign-Urbana.

No words can capture how grateful and indebted I am to my parents for their unconditional faith in me. They were absolutely amazing in supporting me with my somewhat brave decision to change my major from literature to computer science, and have believed in me through every step and turn that I have taken since then. The journey of my Ph.D. would have been much harder if it were not for you.

Last but not least, I would like to thank my husband Joonho for being a wonderful friend, a supportive spouse, an insightful peer researcher, and a loving father to our daughter. You are the only one who knows and understands me in every possible way, and yet never lets go of my hand. I am endlessly grateful for your support and sacrifice for me, and for our memories in Champaign-Urbana. Our daughter Irene, who arrived during the last part of this journey, has been my most precious gift in my life. Thank you for giving me strength and reason to finish this dissertation. I proudly dedicate this dissertation to my husband and my daughter.

Table of Contents

List of Tables . . . . **ix**

List of Figures . . . . **x**

Chapter 1 Introduction **1**
1.1 Contributions 2
1.2 Long-Term Impact 6
1.3 Organization 7

Chapter 2 DeNovoD: Rethinking the Memory Hierarchy for Disciplined Determinism **9**
2.1 Software Assumptions 9
2.1.1 DPJ for Disciplined Determinism 10
2.2 DeNovoD Coherence and Consistency 11
2.2.1 DeNovoD with Equal Address/Communication and Coherence Granularity . . . 12
2.2.2 Conveying and Representing Regions in Hardware 16
2.2.3 DeNovoD with Address/Communication Granularity > Coherence Granularity . 19
2.2.4 Flexible Coherence Granularity 20
2.3 Protocol Optimizations 21
2.3.1 Eliminating Indirection 21
2.3.2 Flexible Communication Granularity 21
2.4 Storage Overhead 22
2.5 Exception Handling 23
2.5.1 Context Switches 24
2.5.2 Thread Migration 25
2.5.3 Error Recovery 26
2.6 Methodology 26
2.6.1 Simulation Environment 26
2.6.2 Simulated Protocols 27
2.6.3 Conveying Regions and Communication Space 28
2.6.4 Protocol Verification 29
2.6.5 Workloads 29
2.7 Evaluation Results 30
2.8 Summary 34

Chapter 3 DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism . . . 35
3.1 Software Assumptions . . . 35
3.1.1 DPJ for Disciplined Non-Determinism . . . 36
3.2 DeNovoND Coherence and Consistency . . . 38
3.2.1 Memory Consistency Model . . . 38
3.2.2 Data Coherence Mechanism . . . 40
3.3 Distributed Queue-Based Locks . . . 41
3.4 Implementation . . . 42
3.4.1 Access Signatures for Coherence of Atomic Accesses . . . 42
3.4.2 Lock Implementation . . . 47
3.4.3 Exception Handling . . . 50
3.5 Overheads . . . 51
3.6 Methodology . . . 52
3.6.1 Simulated Systems . . . 53
3.6.2 Workloads . . . 54
3.7 Evaluation Results . . . 55
3.7.1 Lock-Efficient Applications . . . 55
3.7.2 Lock-Inefficient Applications . . . 58
3.7.3 Optimizations . . . 60
3.8 Summary . . . 62

Chapter 4 DeNovoSync: Supporting Arbitrary Synchronization without Writer-Initiated Invalidation . . . 63
4.1 Software Assumptions . . . 63
4.2 Design Overview . . . 65
4.2.1 The DeNovoSync0 Protocol . . . 65
4.2.2 The DeNovoSync Protocol . . . 69
4.2.3 Increment Counter . . . 70
4.3 Qualitative Analysis . . . 72
4.3.1 Locks . . . 73
4.3.2 Non-Blocking Algorithms . . . 75
4.3.3 Barriers . . . 76
4.4 Methodology . . . 77
4.4.1 Simulated Systems . . . 77
4.4.2 Workloads . . . 77
4.5 Evaluation Results . . . 80
4.5.1 Results for Synchronization Kernels . . . 81
4.5.2 Results for Applications . . . 86
4.6 Summary . . . 87

Chapter 5 Related Work . . . 88
5.1 Software-Hardware Co-designed Systems . . . 88
5.2 Hardware Optimizations for Coherence and Synchronization . . . 91
5.3 GPU Coherence Protocol Designs . . . 92
5.4 Software-Based Coherence . . . 93

5.5 Distributed Shared-Memory 94
5.6 Other Related Work 95

Chapter 6 Conclusions and Future Directions 96
6.1 Conclusions 96
6.2 Open Questions and Future Directions 97
6.2.1 DeNovo Compiler Framework 98
6.2.2 DeNovoAll as a Complete Coherence Solution 100
6.2.3 DeNovoSync as Synchronization Support for Related Systems 101
6.2.4 Protocol-Aware Algorithm Design 101
6.2.5 Support for Commercial Workloads 102
6.2.6 Modeling Energy Efficiency 102
6.2.7 Evaluating the Impact of Context Switches 103

References 104

List of Tables

2.1 Baseline DeNovoD cache coherence protocol for (a) private L1 and (b) shared L2 caches. Self-invalidation and touched bits are not shown here since these are local operations as described in the text. Request buffers (MSHRs) are not shown since they are similar to single core systems. 15
2.2 Simulated system parameters for DeNovoD. 27

3.1 State transitions for a lock word. X indicates unreachable states. 46
3.2 Simulated system parameters for DeNovoND. 52

4.1 Benchmark inputs. 79
4.2 Simulated system parameters for DeNovoSync. 79

List of Figures

1.1 Assumptions and constraints for DeNovoD, DeNovoND, and DeNovoSync. 3

2.1 Software assumptions and execution semantics for DeNovoD. 10
2.2 (a) Code with DeNovoD regions and self-invalidations and (b) cache state after phase 1 self-invalidations and direct cache-to-cache communication with flexible granularity at the beginning of phase 2. X_i represents $S[i].X$. Ci in L2 cache means the word is registered with Core i. Initially, all lines in the caches are in `valid` state. 16
2.3 Comparison of MESI vs. DeNovoD protocols. All bars are normalized to the corresponding ML protocol. 32

3.1 Software assumptions and execution semantics for DeNovoND. 37
3.2 An example of propagating atomic writes using access signatures. Assume a and b are in the same cache line. 44
3.3 Example showing L1 and L2 data layout for the distributed queue-based lock (a) before writeback and (b) after writeback. 49
3.4 Execution time and memory stall time of lock-efficient applications on 6 configurations, normalized to MIL. 55
3.5 Network traffic of all applications on MPL and DQL-256, normalized to MPL. 56
3.6 Total execution time of lock-inefficient applications on six configurations: (a) baseline, (b) with "write-once" atomic region optimization and signature clearing (threshold=99%) applied, normalized to the MESI with idealized locks (MIL) configuration. 57

4.1 Software assumptions and execution semantics for DeNovoSync. 64
4.2 Pseudo code for Michael-Scott queue. 67
4.3 Example interleavings for enqueue() of Michael-Scott queue on MESI, DeNovoSync0, and DeNovoSync. 68
4.4 Test-and-Test-and-Set (TATAS) locks based synchronization. 80
4.5 Array locks based synchronization. 80
4.6 Non-blocking algorithms. 80
4.7 Barrier synchronization (UB = unbalanced computations). 80
4.8 Execution time and network traffic for MESI and DeNovoSync for applications. 86

Chapter 1

Introduction

Shared-memory remains the most widely used programming model among multicore programmers and is the de facto model provided by multicore hardware. Despite the advantage of its global address space, shared-memory programs are known to be difficult to debug and maintain [80], mainly because of unstructured parallel control, data races, and ubiquitous non-determinism. At the same time, providing hardware cache coherence and consistency that can scale in a power-efficient manner to hundreds of cores for such a software model is also a significant challenge. Directory-based cache coherence protocols are notoriously complex to verify [2] and hard to scale, and they remain an active area of research (e.g., [52, 86, 103, 128, 130, 126]). More fundamentally, despite decades of research, it has been difficult to define an acceptable memory semantics (the memory model) for current popular systems, resulting in a call for rethinking current languages and hardware [5].

The DeNovo project takes the view that these problems are not inherent to a global address space paradigm. Instead, they occur due to undisciplined programming models that use arbitrary reads and writes for implicit and unstructured communication and synchronization. Such "wild" shared-memory behaviors undermine the efficiencies that can be achieved through parallelism. These unstructured parallel programming practices not only make it difficult to test and maintain software but also complicate hardware, preventing it from scaling in a performance- and power-efficient manner.

There has been much recent research on "disciplined" shared-memory programming models that address the above software problems with explicit and structured parallelism, synchronization, and communication (e.g., [47, 27, 22, 76, 13, 99, 11, 17, 50, 23, 24]). Although the details vary, they all share the goal of "making parallel programming easier and safer" and expose metadata about program structures and memory access patterns (either automatically extracted or provided by programmers) to prove the desirable properties of programs. The DeNovo project believes that the evolving software landscape represents a unique opportunity for a new multicore architecture paradigm, and asks the following question: *if software becomes more disciplined, can we build a more performance-, power-, and complexity-efficient shared-memory hardware?* The DeNovo project shows that disciplined shared-memory programming models can drive a holistic rethinking of the multicore memory hierarchy.

We apply such a software-driven approach strategically fist to constrained software and then widen the space to include more general software. Figure 1.1 summarizes our approach. The key insight is "separating concerns" for software as shown in the top portion of the diagram. We first distinguish memory accesses based on whether the access is for *data* (non-synchronization) or for *synchronization* (determined by whether it is involved in a race or not [43]). Then, for data accesses, we distinguish between *deterministic* and *non-deterministic* data accesses (defined in Sections 2.1 and 3.1). The distinction between the different access classes comes from knowledge provided by disciplined software. The table in the middle then summarizes how the three DeNovo systems discussed in Chapters 2, 3, and 4 address each access class with a different set of software assumptions, execution semantics, and hardware design decisions.

The strategy not only allows us to isolate what is really important to implement in hardware and what can be obtained from software but also helps to maintain the additional hardware support required minimal for every extension. Therefore, the DeNovo system has evolved from greatly simplifying the hardware by focusing first on deterministic codes to seamlessly adding support for non-deterministic codes and racy synchronization accesses without compromising the performance and simplicity advantages of the original system. Finally, for future work, we propose DeNovoAll which merges all DeNovo systems and presents a comprehensive shared-memory coherence solution for software with different levels of discipline.

1.1 Contributions

In this section, we introduce different DeNovo systems on their evolving paths in terms of the supported software scope and describe the design and contribution of each system in detail.

DeNovoD: Support for Deterministic Codes:[1] There has recently been a surge in software research

[1]I co-led this work with my colleagues, Byn Choi and Rakesh Komuravelli. I contributed to the overall design, implementation, and evaluation of the protocol. This work also appears in Rakesh Komuravelli's Ph.D. thesis.

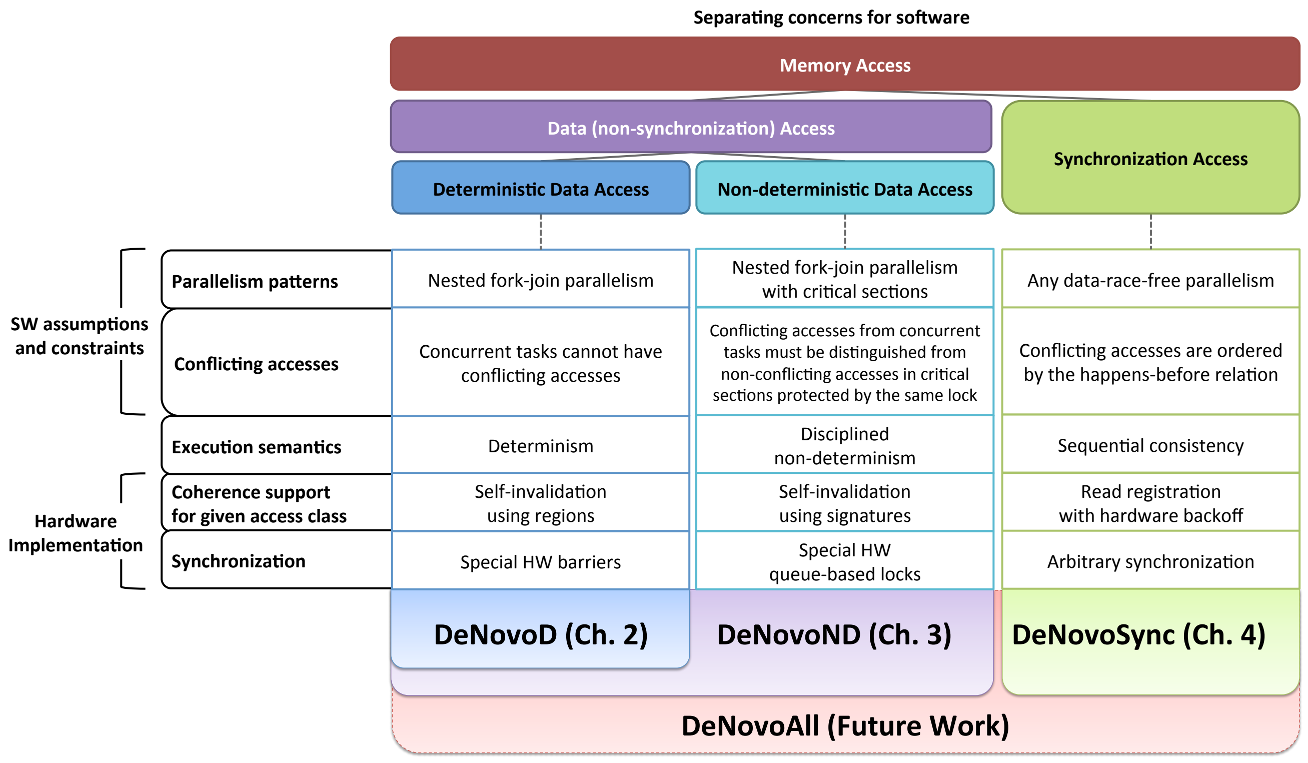

Separating concerns for software
Memory Access
Data (non-synchronization) Access
Synchronization Access
Deterministic Data Access
Non-deterministic Data Access
SW assumptions and constraints
Parallelism patterns
Nested fork-join parallelism
Nested fork-join parallelism with critical sections
Any data-race-free parallelism
Conflicting accesses
Concurrent tasks cannot have conflicting accesses
Conflicting accesses from concurrent tasks must be distinguished from non-conflicting accesses in critical sections protected by the same lock
Conflicting accesses are ordered by the happens-before relation
Execution semantics
Determinism
Disciplined non-determinism
Sequential consistency
Hardware Implementation
Coherence support for given access class
Self-invalidation using regions
Self-invalidation using signatures
Read registration with hardware backoff
Synchronization
Special HW barriers
Special HW queue-based locks
Arbitrary synchronization
DeNovoD (Ch. 2)
DeNovoND (Ch. 3)
DeNovoSync (Ch. 4)
DeNovoAll (Future Work)

Figure 1.1: Assumptions and constraints for DeNovoD, DeNovoND, and DeNovoSync.

to make it easier to debug, test, and maintain parallel programs with more *disciplined* shared-memory programming models [47, 27, 22, 76, 13, 99, 41, 11, 17, 23, 24]. DeNovoD [38] in Chapter 2 [2] concerns the first step of the DeNovo hardware driven by such disciplined programming models. We use Deterministic Parallel Java (DPJ) [23] as an exemplar disciplined language to drive the DeNovoD hardware design. With DeNovoD, we aim to show that shared-memory hardware can be made much simpler and more efficient by exploiting safety guarantees and memory access information provided by disciplined software.

DeNovoD focuses on supporting *deterministic* codes that produce the same external output for a given input. Determinism is a common execution semantics for many computational algorithms, including scientific computing, encryption/decryption, and compiler and program analysis. DeNovoD assumes disciplined deterministic programs with (1) nested fork-join parallelism where concurrent "tasks" are generated at the beginning of a parallel construct and synchronized at an implicit barrier at the end, and (2) no conflicting memory accesses from such concurrent tasks (data-race-freedom). The data-race-

[2]This work was called DeNovo in previous papers, but for clarity, we refer to it as DeNovoD in this thesis and use DeNovo collectively for the whole project and all three systems.

freedom guarantee combined with structured parallel control enables DeNovoD to provide coherence for deterministic memory accesses simply by self-invalidating stale data in caches at explicit synchronization boundaries. To identify which data is potentially stale and needs to be invalidated, DeNovoD assumes that disciplined software provides information on which memory locations will be read or written in a given parallel construct. Such self-invalidations remove the need for a hardware directory to track sharer lists and to send invalidations and acknowledgments on writes. In addition, the data-race-freedom guarantee eliminates conflicting data accesses from concurrent tasks and corresponding transient states in coherence protocols, eliminating a major source of complexity. For synchronization, DeNovoD assumes that barriers are supported with special hardware.

Specifically, DeNovoD's protocol has the following advantages compared to conventional directory-based protocols: (1) the implementation has no transient states and thus is much easier to verify (verification is an order of magnitude faster) and much easier to extend (incorporating optimizations did not introduce any protocol state changes); (2) DeNovoD does not rely on writer-induced invalidations, so it eliminates invalidation message traffic and does not require storage overhead for sharer lists in directories, removing a key source of unscalability; and (3) DeNovoD maintains coherence states at the granularity at which data is shared and thus does not suffer from false sharing (the added state overhead is much smaller than the reduced directory state). Overall, DeNovoD performs about the same as or better than MESI for a range of deterministic applications with up to 77% improved memory stall time and up to 71% reduced network traffic [38], which can be translated into significant energy savings.

DeNovoND: Support for Disciplined Non-Determinism:[3] There are many common codes that are non-deterministic or contain parts that are non-deterministic, most commonly through lock synchronization. For commercial hardware to exploit the benefits of DeNovo, it is imperative that we develop techniques to support non-deterministic codes with performance that at least matches that of conventional systems, without losing the benefits of DeNovoD.

Continuing the software-driven design approach, DeNovoND [119, 120], addressed in in Chapter 3, focuses on extending DeNovoD for deterministic codes to support disciplined non-deterministic codes as well. While DeNovoND still assumes that parallelism is expressed only through fork-join parallel

[3]Although this is joint work with my colleague Rakesh Komuravelli, I led the work and was responsible for the signature-based protocol design, implementation, and evaluation.

constructs, it permits codes to produce non-deterministic output for a given input by allowing conflicting accesses from concurrent tasks in isolated code sections. We assume disciplined languages (with DPJ [24] as an exemplar language), which ensure that potential non-determinism in software is allowed only when it is explicitly requested with proper annotations (more details are given in Chapter 3). DeNovoND assumes (1) a special instruction tag to distinguish non-deterministic accesses from deterministic accesses and (2) critical sections with locks for isolating non-determinism, where the same lock is used to protect conflicting accesses to a given location.

As for the hardware implementation of DeNovoND, we show that modest extensions to DeNovoD can allow this form of non-determinism without sacrificing its advantages. The key insight is to use small and simple hardware Bloom filters as signatures to track and communicate only non-deterministic accesses across explicit lock transfers. The locks themselves are implemented in hardware using ideas similar to distributed queue-based locks or QOSB [48], without requiring sharer's lists and invalidation messages.

Overall, DeNovoND retains the advantages of DeNovoD while significantly expanding the class of programs that it supports without compromising performance. For data accesses, no new externally visible states are added; the only support needed is a signature per core and the ability to transfer it to the next acquirer and use it for self-invalidation at subsequent reads. We continue to have no directories, no invalidations, and no false sharing. DeNovoND provides comparable or better performance than MESI for several applications designed for lock synchronization, and shows 33% less network traffic on average, implying potential energy savings [119].

DeNovoSync: Support for Arbitrary Synchronization Accesses: DeNovoD and DeNovoND focus on providing coherence and consistency for deterministic and non-deterministic data accesses, while assuming barrier synchronization with special hardware support [38] and disciplined locks with a special queue-based lock implementation [119]. However, for widespread adoption, DeNovo hardware must be able to support arbitrary synchronization beyond barriers and disciplined locks. The challenge is that synchronization accesses are *inherently "racy"* and depend on writer-initiated invalidation for a cached reader to see a new value. Unlike race-free data accesses for which DeNovo's reader-initiated self-invalidations work well with known software information and guarantees, racy synchronization accesses do not have any a priori knowledge of when a value may be updated.

In Chapter 4, we propose a simple and efficient mechanism, DeNovoSync, to support synchronization accesses on DeNovo without affecting any of its advantages. DeNovoSync makes one assumption about the software that it obeys the standard data-race-free memory model adopted by C++, Java, and other languages [84, 25] that define sequentially consistent semantics for data-race-free programs. The data-race-free memory model requires a distinction between data and non-data (synchronization) accesses and ensuring that data accesses are ordered by the happens-before relation. The distinction between data and synchronization accesses allows DeNovoSync to focus on providing sequential consistency for synchronization accesses and supporting arbitrary synchronizations, while coherence for data accesses is supported by mechanisms previously proposed for DeNovoD/DeNovoND.

For the sequentially consistent execution of synchronization accesses, we serialize synchronization reads and writes by enforcing both reads and writes to acquire ownership at the directory. For many-reader cases where this simple solution will suffer from ping-ponging the synchronization variable among caches, we use a *hardware backoff* solution that uses coherence protocol transitions to detect when there are concurrent readers and attempts to delay synchronization reads.

The resulting DeNovoSync protocol shows a 26% reduction in execution time and a 50% reduction in network traffic on average, with synchronization microbenchmarks representing a variety of non-blocking data structures and synchronization constructs. For the applications we studied, DeNovoSync provides comparable performance (4% better) and reduces traffic by 26% on average, except for one application [118].

1.2 Long-Term Impact

The global address space provided by the shared-memory programming model simplifies many aspects of parallel programming. Unfortunately, the hardware coherence protocols and consistency models required to provide the illusion of a single global address space are becoming increasingly complex and inefficient. Some researchers have stated that hardware advances built upon (mostly) conventional hardware protocols will be sufficient to meet the complexity-, performance-, and energy-efficiency challenges of future systems. This would certainly have the least disruptive impact on the existing software base. Unfortunately, it is not (yet) clear whether a hardware-only solution will be viable (e.g., [30, 68, 61]).

The DeNovo project adopts a combined hardware-software approach. Such an approach can be risky because it is motivated by a predicted evolution of software. If successful, however, it has the potential for a very high payoff because it holistically rethinks the memory hierarchy from the ground up, driven entirely by software requirements, eliminating needless complexity, performance, and power inefficiencies.

For the DeNovo vision to succeed, the class of programs that can be supported should not be limited to a "disciplined" subset of software. Driven by the key insight regarding the need to separate concerns for racy vs. non-racy accesses and those for deterministic vs. non-deterministic non-racy accesses, our proposed mechanisms successfully reinforce the validity of the approach, and we show its potential for a general-purpose shared-memory solution. We believe that there is significant momentum from the software community to embrace the types of discipline we seek. The standard data-race-free memory model adopted by modern languages [84, 25] guides programmers to write data-race-free programs by providing sequentially consistent semantics only for them. We can run *any* data-race-free software on DeNovoSync and improve performance further with DeNovoD and DeNovoND for disciplined software with metadata.

We do not expect that all software will become disciplined or that legacy codes will not need to be supported. We propose future work to explore an adaptive coherence solution for software with different levels of discipline, including legacy codes.

1.3 Organization

The following chapter proposes and evaluates DeNovoD for disciplined deterministic codes. Chapter 3 proposes and evaluates how DeNovoND extends DeNovoD, described in Chapter 2, to support disciplined non-deterministic codes while maintaining all the benefits of the original DeNovoD. Chapter 4 proposes and evaluates DeNovoSync, an efficient hardware mechanism to support synchronization accesses on DeNovo. DeNovoSync presents a major step toward DeNovo as a comprehensive coherence solution, as it significantly relaxes software requirements for *any* data-race-free software with distinctions for data and synchronization accesses.[4] Chapter 5 discusses previous work that is closely related to the DeNovo

[4]Most of the text in Chapters 2, 3, and 4 is taken from the original publications [38], [119], and [118], respectively.

systems presented in this thesis. Finally, Chapter 6 summarizes the work and outlines future work.

Chapter 2

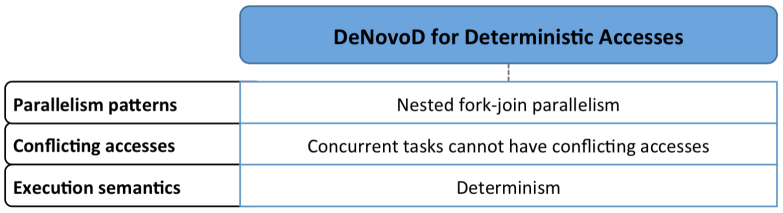
DeNovoD: Rethinking the Memory Hierarchy for Disciplined Determinism

Recent research on disciplined shared-memory programming models has focused on making parallel programming safer and more tractable with explicit and structured communication and synchronization and safety guarantees [47, 27, 22, 105, 76, 13, 99, 11, 17, 50, 23, 24]. We believe that this evolution in the software landscape represents a unique opportunity for a new multicore architecture paradigm. This chapter concerns DeNovoD [38], the first step of the DeNovo hardware project that asks following the question: *if software becomes more disciplined, can we build a more performance-, power-, and complexity-scalable hardware?* In this chapter, we focus on deterministic codes for three reasons. (1) There is a growing view that deterministic algorithms will be common, at least in client-side computing [8]. (2) Focusing on these codes allows us to investigate the "best case"; i.e., the potential gain from exploiting strong discipline. (3) These investigations will form a basis on which we develop the extensions needed for other classes of codes; in particular, disciplined non-determinism and synchronization races as addressed in the following chapters. With DeNovoD, we aim to show that hardware coherence can be made much simpler and more efficient than conventional hardware protocols by exploiting the data-race-freedom guarantee and knowledge about deterministic memory accesses provided by disciplined software.

2.1 Software Assumptions

Figure 2.1 summarizes our software assumptions and execution semantics for disciplined deterministic codes.

1. **Parallelism patterns:** We assume that parallelism is expressed through a nested fork-join structure with an implicit barrier at the join in disciplined software. Such a parallel construct



	DeNovoD for Deterministic Accesses
Parallelism patterns	Nested fork-join parallelism
Conflicting accesses	Concurrent tasks cannot have conflicting accesses
Execution semantics	Determinism

Figure 2.1: Software assumptions and execution semantics for DeNovoD.

divides a program into a series of potentially nested “phases.” We use the term “phase” to refer to the execution of all tasks created by a single parallel construct.

2. **Conflicting accesses:** We assume that there are no conflicting accesses among parallel tasks in a phase; i.e., that the program is data-race-free.

3. **Execution semantics:** Disciplined languages ensure at compile or run time that software obeys the above assumptions about structured parallelism and data-race-freedom. Such software produces *deterministic* results on DeNovoD; i.e., the same externally visible results in all executions with a particular input (Section 2.1.1 provides more details).

2.1.1 DPJ for Disciplined Determinism

As an example of a disciplined language with the above properties, we use Deterministic Parallel Java (DPJ) [23, 24] to drive our software-hardware co-design approach. We assume only type-checked deterministic DPJ programs or its equivalent can run on DeNovoD.

DPJ is an extension to Java that enforces *deterministic-by-default* semantics via compile-time type checking. DPJ provides a new type and effect system for expressing important patterns of deterministic and non-deterministic parallelism in imperative, object-oriented programs. Non-deterministic behavior can only be obtained via certain explicit constructs. For a program that does not use such constructs, DPJ guarantees that if the program is well-typed, any two parallel tasks are *non-interfering*, i.e., do not have conflicting accesses. While [23] allows disciplined behaviors only for deterministic data accesses, [24] later extends [23] to support disciplined non-determinism. In this section, we focus on how [23] guarantees data-race-freedom for deterministic codes first.

DPJ's parallel tasks are iterations of an explicitly parallel `foreach` loop or statements within a `cobegin` block; they synchronize through an implicit barrier at the end of the loop or block. Parallel control flow thus follows a scoped, nested, fork-join structure, which simplifies the use of explicit coherence actions in DeNovoD at fork/join points. This structure defines a natural ordering of the tasks, as well as an obvious definition (omitted here) of when two tasks are "concurrent". It implies an obvious sequential equivalent of the parallel program (`for` replaces `foreach` and `cobegin` is simply ignored). DPJ guarantees that the result of a parallel execution is the same as the sequential equivalent.

In a DPJ program, the programmer assigns every object field or array element to a named "*region*" and annotates every method with read or write "*effects*" summarizing the regions read or written by that method (a region can be non-contiguous in memory). The compiler checks that (i) all program operations are type safe in the region type system; (ii) a method's effect summaries are a superset of the actual effects in the method body; and (iii) that no two parallel statements interfere (conflict). The effect summaries on method interfaces allow all these checks to be performed without interprocedural analysis. A type checked program is guaranteed determinism by default; i.e., the execution appears to occur in the sequential order implied by the program. DPJ [23] has been evaluated on a wide range of deterministic parallel programs. The results show that DPJ can express a wide range of realistic parallel algorithms, and that well-tuned DPJ programs exhibit good performance.

In the following sections, we describe how such safety guarantee, structured parallel control, and the regions and effect information enable the hardware to significantly simplify coherence protocol design.

2.2 DeNovoD Coherence and Consistency

A shared-memory design must first and foremost ensure that a read returns the correct value, where the definition of "correct" comes from the memory consistency model. Modern systems divide this responsibility between two parts: (i) cache coherence, and (ii) various memory ordering constraints. These are arguably among the most complex and hard to scale aspects of shared-memory hierarchy design. Disciplined models enable mechanisms that are potentially simpler and more efficient to achieve this function.

The deterministic parts of our software have semantics corresponding to those of the equivalent sequential program. A read should therefore simply return the value of the last write to the same location that is before it in the deterministic sequential program order. This write either comes from the reader's own task (if such a write exists) or from a task preceding the reader's task, since there can be no conflicting accesses concurrent with the reader (two accesses are concurrent if they are from concurrent tasks). In contrast, conventional (software-oblivious) cache coherence protocols assume that writes and reads to the same location can happen concurrently, resulting in significant complexity and inefficiency.

To describe the DeNovoD protocol, we first assume that the coherence granularity and address/ communication granularity are the same. That is, the data size for which coherence state is maintained is the same as the data size corresponding to an address tag in the cache and the size communicated on a demand miss. This is typically the case for MESI protocols, where the cache line size (e.g., 64 bytes) serves as the address, communication, and coherence granularity. For DeNovoD, the coherence granularity is dictated by the granularity at which data-race-freedom is ensured – a word for our applications. Thus, this assumption constrains the cache line size. We henceforth refer to this as the word based version of our protocol. We relax this assumption in Section 2.2.3, where we decouple the address/communication and coherence granularities and also enable sub-word coherence granularity.

Without loss of generality, throughout we assume private and writeback L1 caches, a shared last-level on-chip L2 cache inclusive of only the modified lines in any L1, a single (multicore) processor chip system, and no task migration. The ideas here extend in an obvious way to deeper hierarchies with multiple private and/or cluster caches and multi-chip multiprocessors, and task migration can be accommodated with appropriate self-invalidations before migration.

2.2.1 DeNovoD with Equal Address/Communication and Coherence Granularity

DeNovoD eliminates the drawbacks of conventional directory protocols as follows.

No directory storage or write invalidation overhead: In conventional directory protocols, a write acquires ownership of a line by invalidating all other copies, to ensure later reads get the updated value. The directory achieves this by tracking all current sharers and invalidating them on a write, incurring significant storage and invalidation traffic overhead. In particular, straightforward bit vector implementations of sharer lists are not scalable. Several techniques have been proposed to reduce this overhead, but typically pay a price in significant increase in complexity and/or incurring unnecessary invalidations when the directory overflows. DeNovoD eliminates these overheads by removing the need for invalidation on a write. Data-race-freedom ensures there is no other writer or reader for that line in this parallel phase. DeNovoD need only ensure that (i) outdated cache copies are invalidated before the next phase, and (ii) readers in later phases know where to get the new data.

For (i), each cache simply uses the known write effects of the current phase to invalidate its outdated data before the next phase begins. The compiler inserts self-invalidation instructions for each region with these write effects (we describe how regions are conveyed and represented below). Each L1 cache invalidates its data that belongs to these regions with the following exception. Any data that the cache has read or written in this phase is known to be up-to-date since there cannot be concurrent writers. We therefore augment each line with a "touched" bit that is set on a read. A self-invalidation instruction does not invalidate a line with a set touched bit or that was last written by this core (indicated by the `registered` state as discussed below); the instruction resets the touched bit in preparation for the next phase.

For (ii), DeNovoD requires that on a write, a core register itself at (i.e., inform) the shared L2 cache. The L2 data banks serve as the registry. An entry in the L2 data bank either keeps the identity of an L1 that has the up-to-date data (`registered` state) or the data itself (`valid` state) – a data bank entry is never required to keep both pieces of information since an L1 cache registers itself in precisely the case where the L2 data bank does not have the up-to-date data. Thus, DeNovoD entails *zero overhead for directory (registry) storage*. Henceforth, we use the term L2 cache and registry interchangeably.

We also note that because the L2 does not need sharer lists, it is natural to not maintain inclusion in the L2 for lines that are not registered by another L1 cache – the registered lines do need space in the L2 to track the L1 id that registered them.

No transient states: The DeNovoD protocol has three states in the L1 and L2 – `registered`, `valid`, and `invalid` – with obvious meaning. (The touched bit mentioned above is local to its cache and irrelevant to external coherence transactions.) Although textbook descriptions of conventional directory protocols also describe 3 to 5 states (e.g., MSI) [56], it is well-known that they contain many hidden transient states due to races, making them notoriously complex and difficult to verify [2, 115, 124]. For example, considering a simple MSI protocol, a cache may request ownership, the directory may forward the request to the current owner, and another cache may request ownership while all of these messages are still outstanding. Proper handling of such a race requires introduction of transient states into the cache and/or directory transition tables.

DeNovoD, in contrast, is a true 3-state protocol with *no transient states*, since it assumes race-free software. The only possible races are related to writebacks. As discussed below, these races either have limited scope or are similar to those that occur in uniprocessors. They can be handled in straightforward ways, without transient protocol states (described below).

The full protocol: Table 2.1 shows the L1 and L2 state transitions and events for the full protocol. Note the lack of transient states in the caches.

Read requests to the L1 (from L1's core) are straightforward – accesses to valid and registered state are hits and accesses to invalid state generate miss requests to the L2. A read miss does not have to leave the L1 cache in a pending or transient state – since there are no concurrent conflicting accesses (and hence no invalidation requests), the L1 state simply stays invalid for the line until the response comes back.

For a write request to the L1, unlike a conventional protocol, there is no need to get a "permission-to-write" since this permission is implicitly given by the software race-free guarantee. If the cache does not already have the line registered, it must issue a registration request to the L2 to notify that it has the current up-to-date copy of the line and set the registry state appropriately. Since there are no races, the write can *immediately* set the state of the cache to registered, without waiting for the registration request to complete. Thus, *there is no transient or pending state for writes either.*

The pending read miss and registration requests are simply monitored in the processor's request buffer, just like those of other reads and writes for a single core system. Thus, although the request buffer technically has transient states, these are not visible to external requests – external requests only see stable cache states. The request buffer also ensures that its core's requests to the same location are serialized to respect uniprocessor data dependencies, similar to a single core implementation (e.g., with MSHRs). The memory model requirements are met by ensuring that all pending requests from the core complete by the end of this parallel phase (or at least before the next conflicting access in the next parallel phase).

The L2 transitions are also straightforward except for writebacks which require some care. A read

	$Read_i$	$Write_i$	$Read_k$	$Register_k$	Response for $Read_i$	Writeback
Invalid	Update tag; Read miss to L2; Writeback if needed	Go to *Registered*; Reply to core i; Register request to L2; Write data; Writeback if needed	Nack to core k	Reply to core k	If tag match, go to $Valid$ and load data; Reply to core i	Ignore
$Valid$	Reply to core i	Go to *Registered*; Reply to core i; Register request to L2	Send data to core k	Go to *Invalid*; Reply to core k	Reply to core i	Ignore
Registered	Reply to core i	Reply to core i	Reply to core k	Go to *Invalid*; Reply to core k	Reply to core i	Go to *Valid*; Writeback

(a) L1 cache of core i. $Read_i$ = read from core i, $Read_k$ = read from another core k (forwarded by the registry).

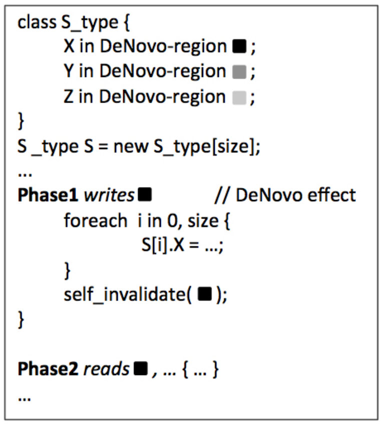
	Read miss from core i	Register request from core i	Read response from memory for core i	Writeback from core i
Invalid	Update tag; Read miss to memory; Writeback if needed	Go to $Registered_i$; Reply to core i; Writeback if needed	If tag match, go to $Valid$ and load data; Send data to core i	Reply to core i; Generate reply for pending writeback to core i
$Valid$	Data to core i	Go to $Registered_i$; Reply to core i	X	X
$Registered_j$	Forward to core j; Done	Forward to core j; Done	X	if i==j, go to $Valid$ and load data; Reply to core i; Cancel any pending Writeback to core i

(b) L2 cache.

Table 2.1: Baseline DeNovoD cache coherence protocol for (a) private L1 and (b) shared L2 caches. Self-invalidation and touched bits are not shown here since these are local operations as described in the text. Request buffers (MSHRs) are not shown since they are similar to single core systems.

or registration request to data that is invalid or valid at the L2 invokes the obvious response. For a request for data that is registered by an L1, the L2 forwards the request to that L1 and updates its registration id if needed. For a forwarded registration request, the L1 always acknowledges the requestor and invalidates its own copy. If the copy is already invalid due to a concurrent writeback by the L1, the L1 simply acknowledges the original requestor and the L2 ensures that the writeback is not accepted (by noting that it is not from the current registrant). For a forwarded read request, the L1 supplies the data if it has it. If it no longer has the data (because it issued a concurrent writeback), then it sends a negative acknowledgment (nack) to the original requestor, which simply resends the request to the L2. Because of race-freedom, there cannot be another concurrent write, and so no other concurrent writeback, to the line. Thus, the nack eventually finds the line in the L2, without danger of any deadlock or livelock. The only somewhat less straightforward interaction is when both the L1 and L2 caches want to writeback the same line concurrently, but this race also occurs in uniprocessors.

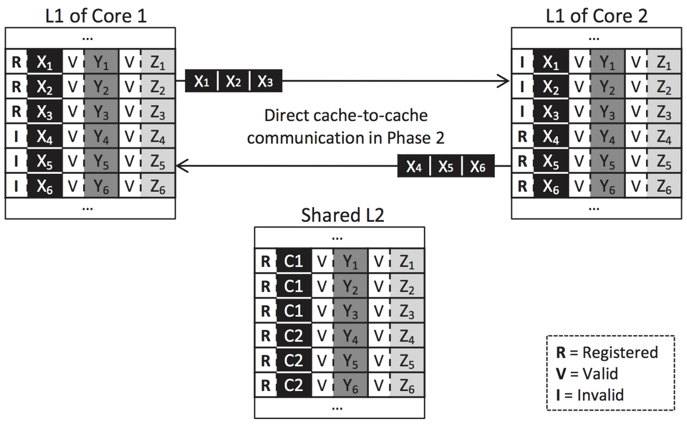
Example: Figure 2.2 illustrates the above concepts. Figure 2.2(a) shows a code fragment with parallel



```
class S_type {
        X in DeNovo-region ■;
        Y in DeNovo-region ▨;
        Z in DeNovo-region □;
}
S _type S = new S_type[size];
...
Phase1 writes ■            // DeNovo effect
        foreach  i in 0, size {
                S[i].X = ...;
        }
        self_invalidate( ■);
}

Phase2 reads ■, ... { ... }
...
```

(a)



(b)

Figure 2.2: (a) Code with DeNovoD regions and self-invalidations and (b) cache state after phase 1 self-invalidations and direct cache-to-cache communication with flexible granularity at the beginning of phase 2. X_i represents $S[i].X$. Ci in L2 cache means the word is registered with Core i. Initially, all lines in the caches are in `valid` state.

phases accessing an array, S, of structs with three fields each, X, Y, and Z. The X (respectively, Y and Z) fields from all array elements form one DeNovoD region. The first phase writes the region of X and self-invalidates that region at the end. Figure 2.2(b) shows, for a two core system, the L1 and L2 cache states at the end of Phase 1, assuming each core computed one contiguous half of the array. The computed X fields are `registered` and the others are invalid in the L1's while the L2 shows all X fields registered to the appropriate cores. (The direct communication is explained in the next section.)

2.2.2 Conveying and Representing Regions in Hardware

A key research question is how to represent regions in hardware. We need to store the region information in the cache at word granularity along with the coherence states for region-based self-invalidations. Language-level regions are usually much more fine-grained than may be practical to support in hardware. For example, when a parallel loop traverses an array of objects, the compiler may need to identify (a field of) *each object* as being in a distinct region in order to prove the absence of conflicts. For the hardware, however, such fine distinctions would be expensive to maintain. Fortunately, we can coarsen language-level regions to a much smaller set without losing functionality in hardware. The key insight is as follows. For self-invalidations, we need regions to identify which data could have been written in the current phase. It is not important to distinguish which core wrote which data. In the above example, we can thus treat the entire array of objects as one region.

Alternately, if only a subset of the fields in each object in the above array is written, then this subset aggregated over all the objects collectively forms a hardware region. Thus, just like software regions, hardware regions need not be contiguous in memory – they are essentially an assignment of a color to each heap location (with orders of magnitude fewer colors in hardware than software). Hardware regions are not restricted to arrays, either. For example, in a traversal of the spatial tree in an n-body problem, the compiler distinguishes different tree nodes (or subsets of their fields) as separate regions; the hardware can treat the entire tree (or a subset of fields in the entire tree) as an aggregate region. Similarly, hardware regions may also combine field regions from different aggregate objects (e.g., fields from an array and a tree may be combined into one region).

The compiler can easily *summarize* program regions into coarser hardware regions as described above and insert appropriate self-invalidation instructions. The only correctness requirement is that the self-invalidated regions must cover all write effects for the phase. For performance, these regions should be as precise as possible. For example, fields that are not accessed or read-only in the phase should not be part of these regions. Similarly, multiple field regions written in a phase may be combined into one hardware region for that phase, but if they are not written together in other phases, they will incur unnecessary invalidations.

During the final code generation, the memory instructions generated can convey the region name of the address being accessed to the hardware; since DPJ regions are parameterizable, the instruction needs to point to a hardware register that is set at runtime (through the compiler) with the actual region number. The above information can be conveyed through one of several techniques. First, existing memory instructions can be extended to include an extra operand for the register number that contains the region ID. Second, ISA with reserve bytes can use them to convey the region register. ISAs that use prefix bytes (e.g., x86) can simply add a new prefix type for memory instructions to indicate that prefix bytes in the given memory instruction carry a region register number and store the register number in the remaining prefix bytes. The address space identifier (ASI) in SPARC ISA can also be also used to convey the region register number. Third, the operating system can augment virtual address spaces with region information. It can provide extended memory allocation interfaces to which programs can pass a region register number. Then the OS encodes the region information in a virtual address (e.g., using upper bits). The OS will return addresses except the region bits, so programs will still see contiguous virtual address spaces, only smaller. On a memory access, the modified address translation logic will locate a full address with the region bits in the page table, read the region register, and convey the region ID to the cache. This scheme does not require ISA extension but needs OS intervention to include the region register number in addresses and pass it down to the hardware. We project fewer than 20 hardware regions per process and even fewer "live" regions that are concurrently accessed in a given phase, based on our evaluation in Section 2.7 (in most cases, a phase has two or three live regions and no more than eight). Since the number of live regions determines the number of region registers, only three additional bits are required to convey the region information. Even if the above extensions are not available and no region information is conveyed to the hardware, DeNovoD can correctly execute programs by being conservative and assuming a single region for all data.

When such augmented memory instructions are executed, they convey the region number to the core's cache. A straightforward approach is to store the region number with the accessed data line in the cache. Then a self-invalidate instruction invalidates all data in the cache with the specified regions that is not `touched` or `registered`.

The above implementation requires storing region bits along with data in the L1 cache and matching region numbers for self-invalidation. A more conservative implementation can reduce this overhead. At the beginning of a phase, the compiler conveys to the hardware the set of regions that need to be invalidated in the *next* phase – this set can be conservative, and in the worst case, represent all regions. Additionally, we replace the region bits in the cache with one bit, `keepValid`, indicating that the corresponding data need not be invalidated until the end of the *next* phase. On a miss, the hardware compares the region for the accessed data (as indicated by the memory instruction) and the regions to be invalidated in the next phase. If there is no match, then `keepValid` is set. At the end of the phase, all data not `touched` or `registered` are invalidated and the `touched` bits reset as before. Further, the identities of the `touched` and `keepValid` bits are swapped for the next phase. This technique allows valid data to stay in the cache through a phase even if it is not `touched` or `registered` in that phase, without keeping track of regions in the cache. The concept can be extended to more than one such phase by adding more bits if the compiler can predict the self-invalidation regions for those phases.

2.2.3 DeNovoD with Address/Communication Granularity > Coherence Granularity

To decouple the address/communication and coherence granularity, our key insight is that any data marked `touched` or `registered` can be copied over to any other cache in `valid` state (but not as `touched`). Additionally, for even further optimization (Section 2.3.1), we make the observation that this transfer can happen without going through the registry/L2 at all (because the registry does not track sharers). Thus, no serialization at a directory is required. When (if) this copy of data is accessed through a demand read, it can be immediately marked `touched`. The above copy does not incur false sharing (nobody loses ownership) and, if the source is the non-home node, it does not require extra hops to a directory.

With the above insight, we can easily enhance the baseline word-based DeNovoD protocol from the previous section to operate on a larger communication and address granularity; e.g., a typical cache line size from conventional protocols. However, we still maintain coherence state at the granularity at which the program guarantees data race freedom; e.g., a word. On a demand request, the cache servicing the request can send an entire cache line worth of data, albeit with some of the data marked invalid (those that it does not have as `touched` or `registered`). The requestor then merges the valid words in the response message (that it does not already have `valid` or `registered`) with its copy of the cache line (if it has one), marking all of those words as `valid` (but not `touched`).

Note that if the L2 has a line `valid` in the cache, then an element of that line can be either `valid` (and hence sent to the requestor) or `registered` (and hence not sent). Thus, for the L2, it suffices to keep just one coherence state bit at the finer (e.g., word) granularity with a line-wide valid bit at the line granularity.[1] As before, the id of the registered core is stored in the data array of the registered location.

This is analogous to sector caches – cache space allocation (i.e., address tags) is at the granularity of a line but there may be some data within the line that is not valid. This combination effectively allows exploiting spatial locality without any false sharing, similar to multiple writer protocols of software distributed shared memory systems [69].

[1]This requires that if a registration request misses in the L2, then the L2 obtain the full line from main memory.

2.2.4 Flexible Coherence Granularity

Although the applications we studied did not have any data races at word granularity, this is not necessarily true of all applications. Data may be shared at byte granularity, and two cores may incur conflicting concurrent accesses to the same word, but for different bytes. A straightforward implementation would require coherence state at the granularity of a byte, which would be significant storage overhead. %footnoteThe C and C++ memory models and the Java memory model do not allow data races at byte granularity; therefore, we also do not consider a coherence granularity lower than that of a byte. Although previous work has suggested using byte based granularity for state bits in other contexts [82], we would like to minimize the overhead.

We focus on the overhead in the L2 cache since it is typically much larger (e.g., 4X to 8X times larger) than the L1. We observe that byte granularity coherence state is needed only if two cores incur conflicting accesses to different bytes in the same word in the same phase. Our approach is to make this an infrequent case, and then handle the case correctly albeit at potentially lower performance.

In disciplined languages, the compiler/runtime can use the region information to allocate tasks to cores so that byte granularity regions are allocated to tasks at word granularities when possible. For cases where the compiler (or programmer) cannot avoid byte granularity data races, we require the compiler to indicate such regions to the hardware. Hardware uses word granularity coherence state. For byte-shared data such as the above, it "clones" the cache line containing it in four places: place i contains the ith byte of each word in the original cache line. If we have at least four way associativity in the L2 cache (usually the case), then we can do the cloning in the same cache set. The tag values for all the clones will be the same but each clone will have a different byte from each word, and each byte will have its own coherence state bit to use (essentially the state bit of the corresponding word in that clone). This allows hardware to pay for coherence state at word granularity while still accommodating byte granularity coherence when needed, albeit with potentially poorer cache utilization in those cases.

2.3 Protocol Optimizations

2.3.1 Eliminating Indirection

Our protocol so far suffers from the fact that even L1 misses that are eventually serviced by another L1 cache (cache-to-cache transfer) must go through the registry/L2 (directory in conventional protocols), incurring an additional latency due to the indirection.

However, as observed in Section 2.2.3, `touched/registered` data can always be transferred for reading without going through the registry/L2. Thus, a reader can send read requests directly to another cache that is predicted to have the data. If the prediction is wrong, a Nack is sent (as usual) and the request reissued as a usual request to the directory. Such a request could be a demand load or it could be a prefetch. Conversely, it could also be a producer-initiated communication or remote write [1, 75]. The prediction could be made in several ways; e.g., through the compiler or through the hardware by keeping track of who serviced the last set of reads to the same region. Owner prediction mechanisms proposed in previous research [3, 66] can be also adapted for DeNovoD to improve prediction efficiency.

The key point is that there is no impact on the coherence protocol – no new states, races, or message types. The requestor simply sends the request to a different supplier. This is in sharp contrast to adding such an enhancement to MESI.

This ability essentially allows DeNovoD to seamlessly integrate a message passing like interaction within its shared-memory model. Figure 2.2 shows such an interaction for our example code.

2.3.2 Flexible Communication Granularity

Cache-line based communication transfers data from a set of contiguous addresses, which is ideal for programs with perfect spatial locality and no false sharing. However, it is common for programs to access only a few data elements from each line, resulting in significant waste. This is particularly common in modern object-oriented programming styles where data structures are often in the form of arrays of structs (AoS) rather than structs of arrays (SoA). It is well-known that converting from AoS to SoA form often gives a significant performance boost due to better spatial locality. Unfortunately, manual conversion is tedious, error-prone, and results in code that is much harder to understand and maintain, while automatic (compiler) conversion is impractical except in limited cases because it requires complex whole-program analysis and transformations [40, 63]. We exploit information about regions to reduce such communication waste, without changing the software's view.

We have knowledge of which regions will be accessed in the current phase. Thus, when servicing a remote read request, a cache could send `touched` or `registered` data only from such regions (recall these are at field granularity within structures), potentially reducing network bandwidth and power. More generally, the compiler may associate a default prefetch granularity attribute with each region that defines the size of each contiguous region element, other regions in the object likely to be accessed along with this region (along with their offset and size), and the number of such elements to transfer at a time. This information can be kept as a table in hardware which is accessed through the region identifier and an entry provides the above information; we call the table the *communication region table*. The information for the table itself may be partly obtained directly through the programmer, deduced by the compiler, or deduced by a runtime tool. Figure 2.2 shows an example of the use of flexible communication granularity – the caches communicate multiple (non-contiguous) fields of region X rather than the contiguous X, Y, and Z regions that would fall in a conventional cache line. Again, in contrast to MESI, the additional support required for this enhancement in DeNovoD does not entail any changes to the coherence protocol states or introduce new protocol races.

This flexible communication granularity coupled with the ability to remove indirection through the registry/L2 (directory) effectively brings the system closer to the efficiency of message passing while still retaining the advantages of a coherent global address space. It combines the benefits of various previously proposed shared-memory techniques such as bulk data transfer, prefetching, and producer-initiated communication, but in a more software-aware fashion that potentially results in a simpler and more effective system.

2.4 Storage Overhead

We next compare the storage overhead of DeNovoD to other common directory configurations.

DeNovoD overhead: At the L1, DeNovoD needs state bits at the word granularity. We have three states and one touched bit (total of 3 bits). We also need region related information. In our applications, we need at most 20 hardware regions – 5 bits. These can be replaced with 1 bit by using the optimization of the `keepValid` bit discussed in Section 2.2.1. Thus, we need a total of 4 to 8 bits per 32 bits or 64 to 128 bits per L1 cache line. At the L2, we just need one valid and one dirty bit per line (per 64 bytes) and one bit per word, for a total of 18 bits per 64 byte L2 cache line or 3.4%. If we assume L2 cache size of 8X that of L1, then the L1 overhead is 1.56% to 3.12% of the L2 cache size.

In-cache full map directory: We conservatively assume 5 bits for protocol state (assuming more than 16 stable+transient states). This gives 5 bits per 64 byte cache line at the L1. With full map directories, each L2 line needs a bit per core for the sharer list. This implies that DeNovoD overhead for just the L2 is better for more than a 13 core system. If the L2 cache size is 8X that of L1, then the total L1+L2 overhead of DeNovoD is better at greater than about 21 (with `keepValid`) to 30 cores.

Duplicate tag directories: L1 tags can be duplicated at the L2 to reduce directory overhead. However, this requires a very high associative lookup; e.g., 64 cores with 4 way L1 requires a 256 way associative lookup. As discussed in [128], this design is not scalable to even low tens of cores system.

Tagless directories and sparse directories: The tagless directories work uses Bloom filter based directory organization [128]. Their directory storage requirement appears to be about 3% to over 5% of L1 storage for core counts ranging from 64 to 1K cores. This does not include any coherence state overhead which we include in our calculation for DeNovoD above. Further, this organization is lossy in that larger core counts require extra invalidations and protocol complexity.

Many sparse directory organizations have been proposed that can drastically cut directory overhead at the cost of sharer list precision, and so come at a significant performance cost especially at higher core counts [128].

2.5 Exception Handling

In this section, we describe how DeNovoD can handle execution exceptions such as context switches, thread migration, and error conditions.

2.5.1 Context Switches

To simplify the simulation environment, we do not measure the performance impact of context switches in our evaluation. To support context switches, DeNovoD must ensure that DeNovo-specific information in hardware is correctly preserved across contexts.

1. **Data with region ID in private caches:** When software issues a store instruction with region information, a region ID is stored with data in a private cache. Depending on whether the cache is physically or virtually tagged/indexed, context switches handle cache contents differently.

   Virtually addressed caches should always invalidate all data to avoid address conflicts on context switches, unless they distinguish different address spaces. With the address space ID (ASID) stored with tags, virtual caches do not need to be entirely invalidated on context switches; DeNovoD can use ASID to perform self-invalidations only on addresses in the current context.

   If caches are physically addressed, cache contents can live across context switches because blocks from different address spaces do not interfere with each other, as their physical addresses are unique. Since the metadata for DeNovoD (e.g., the region ID and touched bit) can be accessed to self-invalidate the memory location associated with it (even without directly addressing the memory), cache contents are no longer context-insensitive for DeNovoD, and leaving them across contexts may affect cache behaviors. Upon executing a self-invalidation instruction, DeNovoD may invalidate blocks from previous contexts with the matching region ID given in an instruction. This may cause unnecessary read misses for those blocks when the context is switched back, but it does not violate correctness by being conservative with respect to data from previous contexts. Similarly, touched bits for data blocks from previous contexts may be reset, which can lead to unnecessary invalidations of valid data.

   Storing the process or thread ID with cache data blocks will prevent valid data from being invalidated in different contexts but will increase storage overheads. Instead, we can add a single "keep" bit per cache line to distinguish between data from previous contexts and current data; this bit is initially unset, and we toggle the bit for all currently valid data in a cache right before a context switch. During active executions, we perform self-invalidation and touched-bit reset only for the data whose keep bit is not set. In this way, we can keep valid data from the previous context untouched in the current context. Evaluating the performance impact of context switches on DeNovoD in detail and exploring mechanisms to minimize it is an important part of our future work.

In summary, it is safe to leave cache contents across contexts for DeNovoD. However, context switches can cause unnecessary invalidations for data from different contexts on DeNovoD unless data is stored in a context-sensitive way as described above. We project that the potential performance impact from context interference will be limited for the following reasons. (1) Context switches are rare events, and the overheads are likely to be amortized over normal execution cycles. (2) When a process is switched out while waiting for synchronization, its non-registered data is self-invalidated and touched bits are reset anyway after the context is restored and synchronization is performed. In this case, context switches do not incur many more "extra" invalidations than the absence of context switches. (3) Cache interference from multiple processes or threads is common [96, 81]. If this hypothesis is true, then data from previous contexts are evicted anyway, and DeNovoD will not see many more read misses from interfering invalidations than other systems.

2. **Pending memory requests:** Before a context switch, pending non-blocking requests in local caches such as write misses (registration) should be completed. This is similar to normal MESI systems that must ensure that all writes complete. It may not be required if we know that the same thread/process will be rescheduled on the same core, but we enforce this on a context switch for simplicity since it is required when a thread gets rescheduled on a different core (i.e., thread migration).

2.5.2 Thread Migration

For context switches, the main issue is how to efficiently "preserve" states of a context-switched thread for when the thread resumes execution on the same core. However, thread migration moves a thread to execute on a different core, abandoning cached data on the old core. Therefore, it is important for DeNovoD to properly invalidate all valid (non-registered) data on the core from which a thread is migrated. We do not need to invalidate registered data because new registrations from a migrated core will take care of the data movement and registration transfer.

2.5.3 Error Recovery

Modern hardware often provides mechanisms for handling and recovering from unexpected errors. One of the common approaches to error detection and recovery involves taking a snapshot of system states on demand or periodically before an error occurs and performing a roll-back to an uncorrupted state on an error. It is a common practice not to include read-only cache contents in a snapshot since they are reproducible and including them will significantly increase the snapshot overheads. Then, the recovered execution will start fresh with read misses for such data. If certain read-only data is critical for error recovery and saved in the snapshot, its region ID must be preserved as well. For registered (modified) data in caches, most error recovery schemes in current use do not inlude it in snapshots by writing it through to the memory before making a snapshot. If a scheme provides snapshots with modified data in the cache, the data along with its region ID should be included in the snapshot, as they are not reproducible from outside a core. Region information for all saved data may make error recovery for DeNovoD more expensive in terms of snapshot storage and latency. However, we believe that this is a very rare event and that the overhead will be amortized over long intervals.

2.6 Methodology

2.6.1 Simulation Environment

Our simulation environment consists of the Simics full-system functional simulator that drives the Wisconsin GEMS memory timing simulator [88] which implements the simulated protocols. We also use the Princeton Garnet [10] interconnection network simulator to accurately model network traffic. We chose not to employ a detailed core timing model due to an already excessive simulation time. Instead, we assume a simple, single-issue, in-order core with blocking loads and 1 CPI for all non-memory instructions. We also assume 1 CPI for all instructions executed in the OS and in synchronization constructs.

Table 2.2 summarizes the key common parameters of our simulated systems. Each core has a 128KB private L1 Dcache (we do not model an Icache). L2 cache is shared and banked (512KB per core). The latencies in Table 2.2 are chosen to be similar to those of Nehalem [51], and then adjusted to take some

Processor Parameters	
Frequency	2GHz
Number of cores	64
Memory Hierarchy Parameters	
L1 (Data cache)	128KB
L2 (16 banks, NUCA)	32MB
Memory	4GB, 4 on-chip controllers
L1 hit latency	1 cycle
L2 hit latency	29 to 61 cycles
Remote L1 hit latency	35 to 83 cycles
Memory latency	197 to 261 cycles

Table 2.2: Simulated system parameters for DeNovoD.

properties of the simulated processor (in-order core, two-level cache) into account.

2.6.2 Simulated Protocols

We compared the following 8 systems:

MESI word (MW) and line (ML): MESI with single-word (4 byte) and 64-byte cache lines, respectively. The original implementation of MESI shipped with GEMS [88] does not support non-blocking stores. Since stores are non-blocking in DeNovoD, we modified the MESI implementation to support non-blocking stores for a fair comparison. Our tests show that MESI with non-blocking stores outperforms the original MESI by 28% to 50% (for different applications).

DeNovoD word (DW) and line (DL): DeNovoD with single-word (Section 2.2) and 64-byte cache lines, respectively. For DL, we do not charge any additional cycles for gathering/scattering valid-only packets. We charge network bandwidth for only the valid part of the cache line plus the valid-word bit vector.

DL with direct cache-to-cache transfer (DD): Line-based DeNovoD with direct cache-to-cache transfer (Section 2.3.1). We use oracular knowledge to determine the cache that has the data. This provides an upper-bound on achievable performance improvement. As discussed in Section 2.3.1, the prediction could be made through different software and/or hardware schemes in real systems. The ideal implementation provides an upper-bound on achievable performance improvement; The efficiency of realistic implementations could be lower than the ideal implementation due to potential prediction failures and retries. However, its impact on performance is limited to DeNovoD with DD only. Designing high-precision prediction schemes is out of the scope of this thesis.

DL with flexible communication granularity (DF): Line-based DeNovoD with flexible communication granularity (Section 2.3.2). Here, on a demand load, the communication region table is indexed by the region of the demand load to obtain the set of addresses that are associated with that load, referred to as the *communication space*. We fix the maximum data communicated to be 64 bytes for DF. If the communication space is smaller than 64 bytes, then we choose the rest of the words from the 64-byte cache line containing the demand load address. We optimistically do not charge any additional cycles for determining the communication space and gathering/scattering that data.

DL and DW with both direct cache-to-cache transfer and flexible communication granularity (DDF and DDFW respectively): Line-based and word-based DeNovoD with the above two optimizations, direct cache-to-cache transfer and flexible communication granularity, combined in the obvious way.

We do not show word based DeNovoD augmented with just direct cache-to-cache transfer or just flexible communication granularity, because the results were as expected and did not lend new insights. Moreover, the DeNovoD word based implementations have too much tag overhead compared to the line based implementations.

2.6.3 Conveying Regions and Communication Space

Regions for self-invalidation: In a real system, the compiler would convey the region of a data through memory instructions (Section 2.2). For this study, we created an API to manually instrument the program to convey this information for every allocated object. This information is maintained in a table in the simulator. At every load or store, the table is queried to find the region for that address (which is then stored with the data in the L1 cache).

Self invalidation: This API call invalidates all the data in the cache associated with the given region, if the data is not `touched` or `registered`. We inserted self-invalidation API calls manually at the end of phases for writeable regions in a given phase. For the applications studied in this paper (see below), the total number of regions ranged from 2 to about 20. These could be coalesced by the compiler, but we did not explore that here.

Communication space: To convey communication granularity information, we again use a special API call that controls the communication region table of the simulator. On a demand load, the table is accessed to determine the communication space of the requested word. In an AoS program, this set can be simply defined by specifying 1) what object fields, and 2) how many objects to include in the set. For six of our benchmarks, these API calls are manually inserted. The seventh, kdTree, is more complex, so we use an automated correlation analysis tool to determine the communication spaces.

2.6.4 Protocol Verification

We used the widely used Murphi model checking tool [42] to formally compare the verification complexity of DeNovoD and MESI. [2] We model checked the word-based protocol of DeNovoD and MESI. We derived the MESI model from the GEMS implementation (the SLICC files) and the DeNovoD model directly from our implementation. To keep the number of explored states tractable, as is common practice, we used a single address / single region (only for DeNovoD), two data values, two cores with private L1 cache and a unified L2 with in-cache directory (for MESI). We modeled an unordered full network with separate request and reply links. Both models allow only one request per L1 in the rest of the memory hierarchy. For DeNovoD, we modeled the data-race-free guarantee by limiting conflicting accesses. We also introduced the notion of phase boundary to provide a realistic model to both protocols by modeling it as a sense reversing barrier. This enables cross phase interactions in both protocols. As we modeled only one address to reduce the number of states explored, we modeled replacements as unconditional events that can be triggered at any time.

2.6.5 Workloads

We use seven benchmarks to evaluate the effectiveness of DeNovoD features for a range of dense-array, array-of-struct, and irregular pointer-based applications. FFT (with input size m=16), LU (with 512x512 array and 16-byte blocks), Radix (with 4M integers and 1024 radix), and Barnes-Hut (16K particles) are from the SPLASH-2 benchmark suite [123]. kdTree [37] is a program for construction of k-D trees which are well studied acceleration data structures for ray tracing in the increasingly important area of graphics and visualization. We run it with the well known bunny input. We use two versions of kdTree: kdTree-false which has false sharing in an auxiliary data structure and kdTree-

[2]Protocol verification is included in this chapter for completeness, but it is not part of my contribution for DeNovoD. This work was separately published by a colleague, Rakesh Komuravelli [73].

padded which uses padding to eliminate this false sharing. We use these two versions to analyze the effect of application-level false sharing on the DeNovoD protocols. We also use fluidanimate (with simmedium input) and bodytrack (with simsmall input) from the PARSEC benchmark suite [19]. To fit into the fork-join programming model, fluidanimate was modified to use the ghost cell pattern instead of mutexes, and radix was modified to perform a parallel prefix with barriers instead of condition variables. For bodytrack, we use its pthread version unmodified.

We chose C/C++ benchmarks to simplify simulation procedures. Without compiler support for conveying software information to the hardware, we manually annalyzed and annotated these benchmarks in the style of DPJ.

2.7 Evaluation Results

We focus our discussion on the time spent on memory stalls and on network traffic since DeNovoD targets these components. Figures 2.3a, 2.3b, and 2.3c respectively show the memory stall time, read miss counts, and network traffic for all eight protocols described in Section 2.6.2 for each application. Each bar (protocol) is normalized to the corresponding (state-of-the-art) MESI-line (ML) bar.

The memory stall time bars (Figure 2.3a) are divided into four components. The bottommost indicates time spent by a memory instruction stalled due to a blocked L1 cache related resource (e.g., the 64 entry buffer for non-blocking stores is full). The upper three indicate additional time spent stalled on an L1 miss that gets resolved at the L2, a remote L1 cache, or main memory respectively. The miss count bars (Figure 2.3b) are divided analogously. The network traffic bars (Figure 2.3c) show the number of flit crossings through on-chip network routers due to reads, writes, writebacks, and invalidations respectively.

For reference, Figure 2.3d shows the overall execution time for all the protocols and applications, divided into time spent in compute cycles, memory stalls, and synchronization stalls respectively.

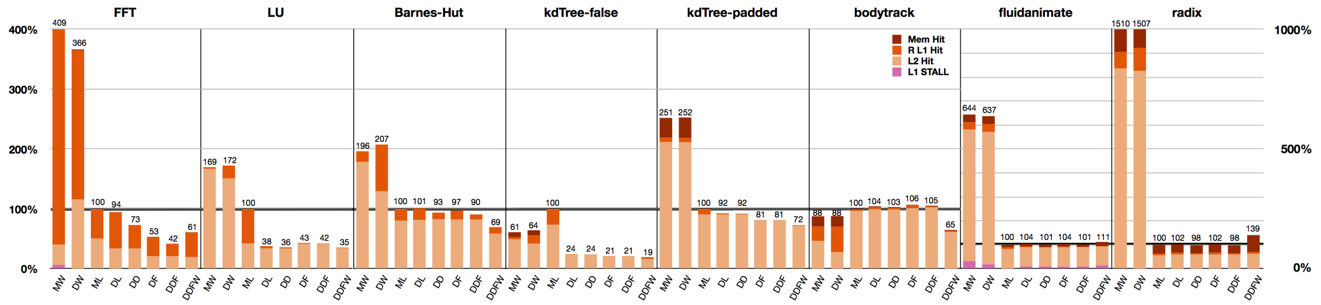
LU and bodytrack show considerably large synchronization times. LU has inherent load imbalance. Using larger input sizes would reduce synchronization time, but prohibitively long simulation times made that impractical for this paper. Bodytrack has several sequential phases and a limited amount of parallelism for the input used (only up to 60 threads in some phases [20]). The idle cores in these phases result in the high synchronization time.

MESI vs. DeNovoD word protocols (MW vs. DW): MW and DW are not practical protocols because of their excessive tag overhead. A comparison is instructive, however, to understand the efficacy of selective self-invalidation, independent of line-based effects such as false sharing. In all cases, DW's performance is competitive with MW. For the cases where it is slightly worse (LU, Barnes and Bodytrack), the cause is higher remote L1 hits in DW than in MW. This is because in MW, the first reader forces the last writer to writeback to L2. Thus, subsequent readers get their data from L2 for MW but need to go to the remote L1 (via L2) for DW, slightly increasing the memory stall time for DW. However, in terms of network traffic, DW always significantly outperforms MW.

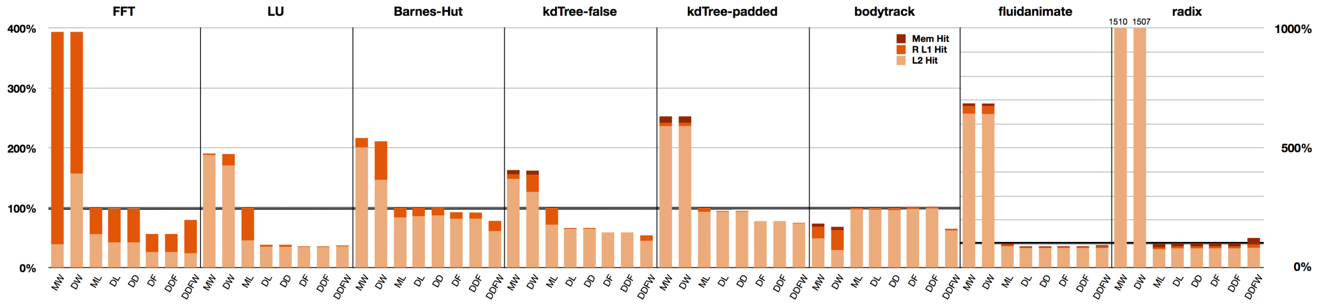
MESI vs. DeNovoD line protocols (ML vs. DL): DL shows about the same or better memory stall times as ML. For LU and kdTree-false, DL shows 62% and 76% reduction in memory stall time over ML, respectively. Here, DL enjoys one major advantage over ML: DL incurs no false sharing due to its per-word coherence state. Both LU and kdTree-false contain some false sharing, as indicated by the significantly higher remote L1 hit component in the miss rate count and memory stall time graphs for ML. In terms of network traffic, DL outperforms ML except for fluidanimate and radix. Here, DL incurs more network traffic because registration (write-traffic) is still at word-granularity (shown in 2.3c). This can be potentially mitigated with a "write-combining" optimization that aggregates individual registration requests similar to a combining write buffer.

Effectiveness of cache lines for MESI: Comparing MW and ML, we see that the memory stall time reduction resulting from transferring a contiguous cache line instead of just a word is highly application dependent. The reduction is largest for radix (a large 93%), which has dense arrays and no false sharing. Most interestingly, for kdTree-false (object-oriented AoS style with false sharing), the word based MESI does better than the line based MESI by 39%. This is due to the combination of false sharing and less than perfect spatial locality. Bodytrack is similar in that it exhibits little spatial locality due to its irregular access pattern. Consequently, ML shows higher miss counts and memory stall times than MW (due to cache pollution from the useless words in a cache line).

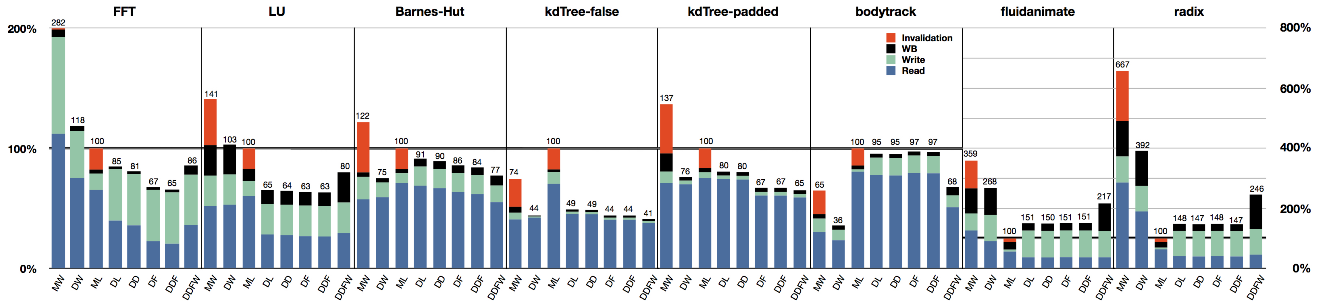
Effectiveness of cache lines for DeNovoD: Comparing DW with DL, we see again the strong application dependence of the effectiveness of cache lines. However, because false sharing is not an issue with DeNovoD, both LU and kdTree-false enjoy larger benefits from cache lines than in the case of



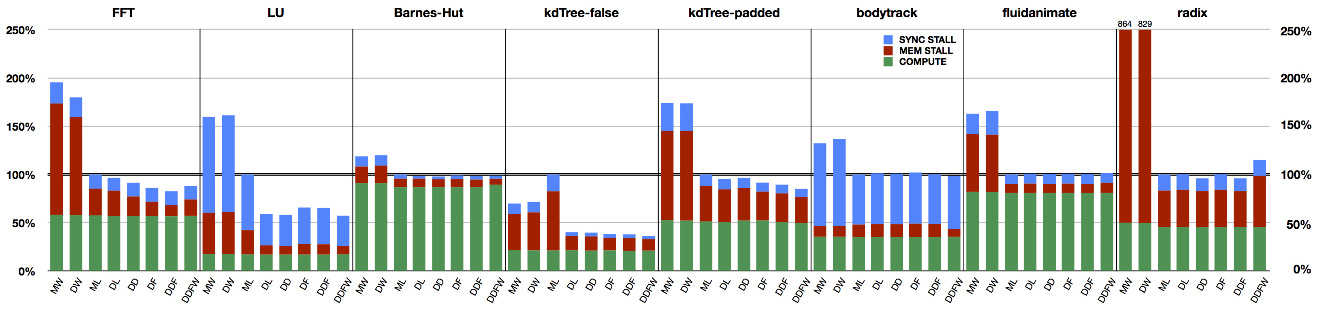
(a) Memory stall time.



(b) Read miss counts.



(c) Network traffic (flit-crossings).



(d) Execution time.

Figure 2.3: Comparison of MESI vs. DeNovoD protocols. All bars are normalized to the corresponding ML protocol.

MESI (78% and 63% reduction in memory stalls). Analogous to MESI, Bodytrack sees larger memory stalls with DL than with DW because of little spatial locality.

Effectiveness of direct cache-to-cache transfer with DL: FFT and barnes exhibit much opportunity for direct cache-to-cache transfer. For these applications, DD is able to significantly reduce the remote L1 hit latencies when compared to DL.

Effectiveness of flexible communication granularity with DL: DF performs about as well or better than ML and DL for all cases, except for LU. LU does not do as well because of the line granularity for cache allocation (addresses). DF can bring in data from multiple cache lines; although this data is likely to be useful, it can potentially replace a lot of allocated data. Bodytrack shows a similar phenomenon, although to a much lesser extent. As we see later, flexible communication at word address granularity does much better for LU and Bodytrack. Overall, DF shows up to 79% reduction in memory stall time over ML and up to 44% over DL. These results are pessimistic since we did not transfer more than 64 bytes of data at a time.

Effectiveness of combined optimizations with DL: DDF combines the benefits of both DD and DF to show either about the same or better performance than all the other line based protocols (except for LU for reasons described above).

Effectiveness of combined optimizations with DW: For applications like LU and bodytrack with low spatial locality, word-based protocols have the advantage over line based protocols by not bringing in potentially useless data and/or not replacing potentially useless data. We find that DW with our two optimizations (DDFW) does indeed perform better than DDF for these two applications. In fact, DDFW does better for 5 out of the 8 applications. This motivates our future work on using a more software-aware (region based) address granularity to get the best benefit of our optimizations.

Effectiveness of regions and touched bits: To evaluate the effectiveness of regions and touched bits, we ran DL without them. This resulted in all the valid words in the cache being invalidated by the self-invalidation instruction. Our results (not shown in detail) show 0% to 25% degradation for different applications, which indicates that these techniques are beneficial for some applications.

Protocol verification results: Through model checking, we found three bugs in DeNovoD and six bugs including two deadlock scenarios in MESI. Note that DeNovoD is much less mature than the GEMS MESI protocol which has been used by many researchers. In DeNovoD, all bugs were simple to fix and showed mistakes in translating our internal high level specification into the implementation (i.e., their solutions were already present in our internal high level description of the protocol). In MESI, all the bugs except one of the deadlocks are caused by protocol races between L1 writebacks and other cache events. These involved subtle races and took several days to track, debug and fix. After fixing all the bugs, the model for MESI explores 1,257,500 states in 173 seconds whereas the model for DeNovoD explores 85,012 states in 8.66 seconds. Our experience clearly indicates the simplicity and reduced verification overhead for DeNovoD compared to MESI.

2.8 Summary

This chapter concerns the first step of the DeNovo hardware project to rethink multi-core memory hierarchies driven by disciplined software models. The key observation is that disciplined programming models will be essential for software programmability and clearly specifiable hardware/software semantics, and can drive a holistic co-design of hardware.

DeNovoD shows that race-freedom, structured parallel control, and the knowledge of regions and effects in deterministic codes enable much simpler, more extensible, and more efficient cache coherence protocols than the state-of-the-art. The resulting protocol has no transient states, no invalidation message traffic, no sharer lists in directories, and no false sharing. A holistic co-design of software and hardware also allows new ideas (e.g., flexible cache partitions based on software specified regions), simpler and more efficient incarnations of previous ideas (e.g., use of bulk transfer, but with flexible software-driven granularity and with no directory serialization), and a synergistic collection of previously proposed optimizations.

Overall, compared to state-of-the-art MESI protocols, DeNovoD is much simpler and easier to verify and extend, performs comparably or better, and is more energy-efficient (since it reduces cache misses and network traffic) for a range of deterministic codes.

Chapter 3

DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism

As explained in the previous chapter, DeNovoD presents a complexity-, performance-, and power-efficient hardware coherence solution for deterministic codes, driven by disciplined programming models. Although determinism is considered desirable for many application classes, many common algorithms take any of multiple possible outputs as legitimate for a given input. Such potential non-determinism in output often allows the algorithms to be more flexible and simpler than deterministic versions. For industry to exploit the benefits of DeNovo, it is imperative that we develop techniques to support non-deterministic codes that perform at least as well as conventional systems, without losing the benefits of DeNovoD.

In this chapter, we propose DeNovoND, a significant step toward achieving the DeNovo vision, by providing support for programs with disciplined non-determinism. We continue to apply our hardware-software co-design approach for DeNovoND; we exploit disciplined programming models with support for safe non-determinism to extend DeNovoD for programs that contain non-determinism. We aim to show that such programs can be supported by simple additions to DeNovoD, without sacrificing DeNovoD's advantages.

3.1 Software Assumptions

In this thesis, we define non-determinism as potential "output non-determinism" through different schedule-dependent interleavings of shared data accesses. To include a larger range of programs in the non-determinism category, we define output as either intermediate or final output. (A program with non-deterministic intermediate output but with deterministic final output is also non-deterministic.) What is important is that the non-determinism should not simply come from uncontrolled and unexpected behavior due to data races. Such behavior is not only potentially erroneous but can also make program executions difficult to maintain and reason about. Therefore, in this chapter, we assume "disciplined non-determinism" for DeNovoND where disciplined languages provide safer and more structured mechanisms to express non-determinism.

Figure 3.1 summarizes the software assumptions and constraints for disciplined non-deterministic codes.

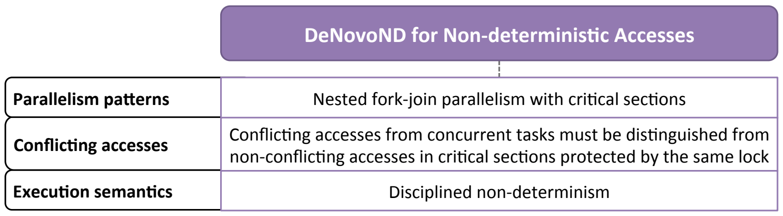
1. **Parallelism patterns:** For parallelism patterns, we assume the same nested fork-join parallelism for disciplined determinism for DeNovoND as for DeNovoD in Section 2.1. Unlike programs with deterministic data accesses only, however, non-deterministic programs can include critical sections with locks in parallel forked tasks. Parallel forks are labeled non-deterministic if their tasks have such critical sections.

2. **Conflicting accesses:** In addition to non-conflicting deterministic accesses as assumed and supported by DeNovoD, conflicting accesses with potential non-determinism are supported by DeNovoND with the following assumptions: (1) non-deterministic accesses are distinguished from deterministic accesses at compile time, and the distinction is conveyed to the hardware. (2) Disciplined non-deterministic accesses to a given memory location are protected by the same lock and enclosed in its critical sections. Non-determinism is allowed only when it is explicitly requested through the aforementioned language constructs. As a result, the program is guaranteed to be data-race-free.

3. **Execution semantics:** Disciplined languages ensure that a potentially non-deterministic program obeys the above properties (structured parallelism and data-race-freedom). Such programs produce sequentially consistent results on DeNovoND with safety guarantees such as strong isolation between tasks within a deterministic fork and non-deterministic tasks with critical sections (refer to Section 3.1.1 for more details).

3.1.1 DPJ for Disciplined Non-Determinism

DeNovoND uses DPJ for non-deterministic codes [24] as an exemplar disciplined programming model to drive its detailed design. To enable the programmer to express non-determinism, DPJ provides parallel



	DeNovoND for Non-deterministic Accesses
Parallelism patterns	Nested fork-join parallelism with critical sections
Conflicting accesses	Conflicting accesses from concurrent tasks must be distinguished from non-conflicting accesses in critical sections protected by the same lock
Execution semantics	Disciplined non-determinism

Figure 3.1: Software assumptions and execution semantics for DeNovoND.

constructs that are potentially non-deterministic; i.e., *foreach_nd* and *cobegin_nd* [24]. These constructs allow conflicting accesses between their tasks, but require that such accesses be enclosed within atomic sections, that their read and write effect declarations also include the atomic keyword, and that their region types be declared as atomic. Note that there continue to be no conflicts allowed between a task from a deterministic parallel construct and any other concurrent (non-deterministic or deterministic) task. The compiler checks that all of the above constraints are satisfied by any type-checked program, again using a simple, modular type checking algorithm.

With the above constraints, DPJ can provide the following guarantees: (1) Data-race freedom. (2) Strong isolation of accesses in atomic section constructs and all deterministic parallel constructs; i.e., these constructs appear to execute atomically. (3) Sequential composition for deterministic constructs; i.e., tasks of a deterministic construct appear to occur in the sequential order implied by the program (even if they contain or are contained within non-deterministic constructs). (4) Determinism-by-default; i.e., any parallel construct that does not contain an explicit non-deterministic construct provides deterministic heap output for a given heap input. The above guarantees not only ensure sequential consistency but also allow programmers to reason with very high-level, strongly isolated, and composable components such as complete *foreach* constructs and all atomic sections.

For data accesses, we assume that the ISA provides a mechanism by which loads and stores can be tagged as accessing atomic regions with atomic effects (e.g., with a bit in the op-code). The DPJ compiler has this information and can generate code with the bit set for such accesses. We refer to such accesses as *atomic* accesses and to others as *non-atomic* accesses. Note that the former are regular data accesses from atomic sections and are not to be confused with atomic read-modify-writes or the C++ atomic keyword used for synchronization races. Support for atomic read-modify-writes in synchronization races is introduced in Chapter 4.

Although DPJ supports atomic sections, DeNovoND assumes we can convert them to locks. This is possible because by default we can associate each atomic region with its own lock. For each atomic section, we can acquire locks for each atomic region that it accesses in a predefined order. This can be optimized in several ways; e.g., by coarsening the locks. An implementation of this algorithm is outside the scope of the thesis. The benchmarks evaluated for DeNovoND in this chapter are either originally written with lock synchronization or manually analyzed and converted to critical sections (from transactions).

3.2 DeNovoND Coherence and Consistency

For deterministic programs, DeNovoD achieves its benefits primarily by replacing writer-induced invalidations (and eliminating related overheads) with compiler-inserted self-invalidations for all writeable data in a given parallel phase (e.g., DPJ's foreach or cobegin construct). DPJ's data-race-freedom guarantee ensures that only the writing core will read that data in a phase, and that all subsequent data reads (in later phases) will see up-to-date values.

However, we cannot assume that a parallel phase will have no conflicting data accesses among concurrent tasks any more for programs with non-deterministic codes. By exploiting DPJ support for disciplined non-determinism [24], DeNovoND [119] knows that such accesses will be protected by the same lock (this lock may change in a different parallel phase), and that such accesses are explicitly identified as atomic accesses in DPJ programs. DeNovoND provides hardware coherence and consistency for these atomic accesses enabled by the software information, as described in detail below.

3.2.1 Memory Consistency Model

For a correct design, we must first understand the constraints imposed by the memory consistency model which specifies what value a read must return.

Informal model: DPJ provides a very strong consistency model. It guarantees sequential consistency and hence a total order over all memory operations (that is consistent with program order). A read must return the value of the last write to its location as defined by this total order. DPJ also enforces additional rules that further constrain this last write for data operations, simplifying reasoning for software and implementation for hardware as follows.

Non-atomic accesses: DPJ ensures that for a non-atomic access, there cannot be a conflicting access by another concurrent task in the same phase. Thus, for a non-atomic read, the last conflicting write is either from its own task or from a task in a previous phase. This is identical to DeNovoD and we can use the identical implementation.

Atomic accesses: For atomic accesses as defined above, DPJ allows conflicting accesses among concurrent tasks, but ensures that all such accesses to a given location are in critical sections protected with the same lock. These critical sections must execute atomically, imposing a total order on all conflicting atomic accesses within a phase. A read therefore must return the value from the (unique) last conflicting write from a critical section in the current phase; if such a write does not exist, then the read must return the (unique) last conflicting write from the previous phase.

Formal model: We now state the model more formally. Note that this model is motivated as a specification for hardware and is therefore at a low level, in terms of individual reads and writes. DPJ programmers work at a higher level in terms of composition and serialization of higher level constructs (cobegin, atomic section, etc.) as described in Section 3.1.1. Our model can be stated in two parts for synchronization and data accesses respectively:

(1) Synchronization accesses are sequentially consistent. This implies a total order between phases and between critical sections to a given lock variable within a phase; this total order is consistent with program order.

(2) For conflicting data accesses, X and Y, we define a *happens-before* relation, denoted $\rightarrow_{hb}$ such that $X \rightarrow_{hb} Y$ iff

- *Type 1 edge:* X's phase precedes Y's phase (by the total order in (1)), or
- *Type 2 edge:* X and Y are in the same task, and X is before Y by program order, or
- *Type 3 edge:* X and Y are atomic accesses in critical sections protected by the same lock variable, and X's critical section precedes Y's critical section (by the total order in (1)).

Then DPJ's guarantees ensure that $\rightarrow_{hb}$ orders all conflicting accesses, and hardware should ensure that a data read returns the value of the last conflicting write in $\rightarrow_{hb}$ order. For a non-atomic read, the last write is always ordered before it by a *type 1* or *type 2* $\rightarrow_{hb}$ edge. For an atomic read, the last write may be ordered before it by a *type 2* or *type 3* edge if such a write exists; otherwise, it is ordered by a *type 1* edge.

3.2.2 Data Coherence Mechanism

The coherence mechanism must simply ensure that a read returns the value from the write as defined by the consistency model. As with DeNovo, we divide the coherence mechanism into two components:

(1) *No stale data:* A read should never see *non-last* (stale) data in its L1 cache(s).

(2) *Locatable up-to-date data:* When a read misses in its L1 cache(s), it should know where to get the *last* (up-to-date) copy of the data.

Above, *last* is precisely defined by the happens-before order. For non-atomic accesses, both components above remain identical to DeNovoD since the consistency model requirements are identical. For atomic accesses, the requirements are met as follows.

No stale data: For the first requirement of no stale data, we use self-invalidations as with DeNovo, thereby precluding the need for adding invalidation messages and directories with sharer lists. Additional self-invalidations are needed with DeNovoND only if there are conflicting atomic accesses among concurrent tasks in a phase (otherwise, DeNovo's self-invalidations at the start of a phase suffice). In the case of conflicting atomic accesses among concurrent tasks, we use the happens-before relation to determine *when* and *what* to self-invalidate as follows.

To determine when to self-invalidate, we note that a concurrent conflicting read must be in a critical section itself and must return the value of the last write also in a critical section protected by the same lock in the same phase (type 2 or 3 edge). Thus, it is sufficient to self-invalidate any time between the start of a critical section and an atomic read in that section.

To determine what to self-invalidate, we have several choices. We could invalidate the entire cache (which seems excessive) or only the atomic regions (for which we would need to keep extra state to identify in the cache). An alternative is for each core to update a signature that records all writes to atomic regions, and then to transfer this signature when the lock is acquired by another core. On a first atomic read to a location, the acquiring core needs to check the signature and self-invalidate the location if it is present in the signature. The acquiring core must forward the union of its signature and the signatures it has received to the next acquirer.

Locatable up-to-date data: For the second requirement of finding the value of the last write on a miss, we use ideas similar to DeNovo. On a write to valid or invalid data, the L1 cache sends a registration request to the L2. The registrations are required to complete before the lock release so that conflicting writes from critical sections are serialized in the right order (it is possible to postpone the registration completion until the next lock acquire). A read that misses in the cache simply goes to the registry (L2) to find the up-to-date value.

Thus we continue with only three states in the protocol as before: *Valid*, *Invalid*, and *Registered*. The extra work over DeNovoD is to update the signature on atomic writes, send the signature on a lock transfer, and invalidate appropriately on atomic reads. Section 3.4.1 discusses each of these steps in more detail.

3.3 Distributed Queue-Based Locks

Our distributed queue-based lock design is modeled after QOSB [48, 64], where the identities of the cores waiting for a lock are maintained in a queue of pointers distributed across the waiting cores' L1 caches and the L2 cache. All requests to a given lock are serialized at the corresponding shared L2 cache bank. The data portion of the L2 cache entry for a contended lock tracks the last requestor (i.e., the tail of the queue of waiters), referred to as *tailPtr*. When the L2 receives the next request for the lock, it forwards it to the current tail's L1. On receiving such a forwarded request, the L1 checks a bit in its copy of the lock word, called the *Locked* bit, to determine if the lock is still held or was unlocked. In the former case, the L1 stores the requestor's ID in another field of the lock word, referred to as *nextPtr*. In the latter case, the L1 responds to the requestor with its signature and transfers the lock, marking its own lock word *Invalid*. When a core releases a lock, its L1 checks its *nextPtr* – if not null, it transfers the lock (with the signature) to the *nextPtr* core; otherwise, it unsets its *Locked* bit. We allow eviction of lock words from the L1 and L2 caches by reusing the data portion of the lock words in the next level of the memory hierarchy to store lock queue information. This approach relies on using L2 data banks to store (non-data) metadata, which is similar to DeNovo's tracking of registration information for the *Registered* state.

3.4 Implementation

This section discusses in detail how DeNovoND implements the memory consistency model and the coherence mechanism described in Section 3.2 using *access signatures* and the distributed queue-based lock mechanism. We also qualitatively discuss the hardware and performance overheads of the implementation.

3.4.1 Access Signatures for Coherence of Atomic Accesses

DeNovoND's memory consistency model requires that a read return the value of the last write preceding it, as ordered by the three types of *happens-before* edges described in Section 3.2. DeNovoD already guarantees that a write ordered by a type 1 or type 2 edge is seen at a read (the former through self-invalidations at the start of a new phase and the latter through single core semantics). For a non-atomic read, a write is ordered only through the above two edge types; therefore, DeNovoD already provides consistency for such reads. For atomic reads where a previous (atomic) write is ordered by a type 3 edge, however, DeNovoND must provide a new mechanism – it needs to track which data in atomic regions has been modified in a critical section in the current phase, as well as a mechanism to efficiently represent and transfer this information on a successful lock acquire.

We use an "access signature" for the purpose of tracking atomic writes. A signature is a compact representation of a set at the expense of precision. Its main functionality includes element insertion, membership query, and flash clear functions. DeNovoND implements the access signature as a small Bloom filter in hardware [21]. Due to its storage efficiency, simplicity, and low access latency, a hardware Bloom filter has been a popular solution for many areas including networking and transactional memory [26, 31].

For our Bloom filters, the keys are addresses accessed (i.e., atomic regions that have atomic effects in this phase), since we are interested only in modifications made to those addresses. The key domain dynamically changes between cores and phases, as a new set of atomic accesses occurs. To keep the false positive rate of Bloom filter reasonably low, the size of each Bloom filter should be determined based on the average size of the key domain. This turns out to be quite small in our case (256 bits suffice) since we only track atomic accesses in a given phase (later sections discuss the size in more detail). We conservatively keep one filter per core to track all modifications across different critical sections (with different locks) on the same core. Thus, for a system with n cores, we have a total of n Bloom filters in the system.

The following uses Figure 3.2 as a running example to show how DeNovoND uses the Bloom filters. On the left, the figure shows DPJ style code depicting three variables, a, b, and c in atomic region xR. It then shows a critical section protected by lock x with atomic read and write effects on region xR. The right side of the figure shows an execution with two cores, *C1* and *C2*. *C2* acquires the lock for the critical section first, followed by *C1* and then *C2* again. The figure also shows the signatures at each core, assuming a perfect hash function.

On atomic writes: An atomic write (as determined by the op-code of the store instruction as discussed in Section 3.2) invokes the same cache protocol operations as in DeNovo. That is, if the word is not in *Registered* state at the L1, a registration request is sent to the L2. Additionally, the word is updated right away and any required writeback is sent to the L2 as well.

For DeNovoND, an atomic write additionally inserts the accessed address into its core's Bloom filter. To avoid repeating insertion of the same address to the Bloom filter, we can add an additional bit, called the "dirty bit," to mark a memory location already updated in a given phase. The "dirty bit" is set on the first atomic store request to a word in a phase, and all dirty bits get unset at the end of a phase. If a store finds the dirty bit already set, it means the word is already inserted into the core's Bloom filter and does not need to be inserted again. Since this is purely an optimization, we can piggyback the functionality of a dirty bit on other state bits described below (e.g., the *touched-atomic* bit) – this may result in some extraneous resets, but does not affect correctness and reduces extra state.

Thus, at the end of a critical section, all addresses modified in the section are recorded in the core's filter; i.e., their entries are non-zero. From Figure 3.2, every store request to a, b, and c in the lightly shaded critical sections updates the Bloom filter on *C1* and *C2*. The second critical section phase on *C2* does not update the Bloom filter since it does not have atomic writes.

On acquire/release: On an acquire, all modifications preceding the release associated with the acquire

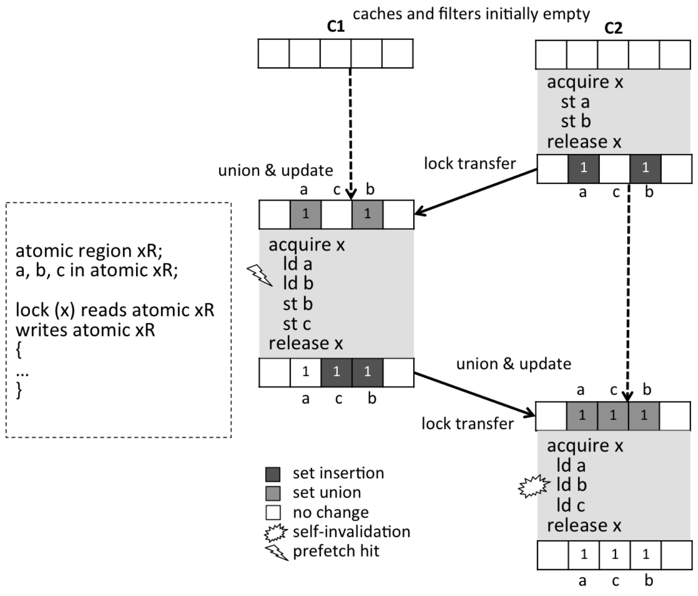


Figure 3.2: An example of propagating atomic writes using access signatures. Assume *a* and *b* are in the same cache line.

are made visible to the acquirer by transferring the access signature at the releaser. The releaser compresses and sends the Bloom filter at its core to the acquirer, when transferring the lock. The acquirer, on receiving the Bloom filter, updates its own Bloom filter by making a *union* of its local Bloom filter and the releaser's Bloom filter. Figure 3.2 shows the resulting Bloom filters at the beginning of each critical section, of which the lightly shaded entries come from the union operation. Note that we only send the signature, not the actual data. On acquire and release points, we also reset the "touched-atomic" and "prefetch" bits (as will be explained in detail below).

On atomic reads: Atomic reads need to conceptually consult the signatures obtained from remote releasers to determine if cached data is valid or stale. If the read is to a word in *Registered* state in the L1, then regardless of the signature state, the word is up-to-date in the cache and the read is a cache hit. If the word is *Invalid* in L1, then a normal read request is sent to L2. If the word is in *Valid* state, then it is also up-to-date if its address does not appear in the access signature. If the word is in *Valid* state and its address hits in the access signature, then it may or may not be up-to-date depending on whether it has been previously read in this critical section.

Specifically, if the word has already been read in this critical section, the previous read brought up-to-date data that is still valid (since no other core can write to the word during the same critical section). We identify this situation by using a *touched-atomic* bit that is set on the first read of the word in a critical section and reset at the release – more precisely, it needs to be reset only when the lock is handed off for another core's acquire (lock hand-off). Thus, a read to a word in *Valid* state with *touched-atomic* bit set is a cache hit.

Another case where a valid word may be up-to-date is when it is obtained as part of a cache line transfer for a demand access to another word in that line. We would like to take advantage of such a prefetch as with conventional cache lines and with DeNovo. If the word comes directly from the L2 or from memory, then it is definitely valid. If it comes from a remote cache, then it is valid if that word was marked as *touched-atomic* or *Registered* in the remote cache. In this case, we can conceptually add another bit called the "prefetch bit" which can be set for prefetched words with the above properties. These bits must be reset on the next lock hand-off or the next acquire, whichever happens first. A read that accesses a valid word with *prefetch* bit set is considered a cache hit. Although the *touched-atomic* and *prefetch* bits are separately motivated, both functions can be achieved by a single bit that we collectively refer to as the *touched-atomic* bit.

In summary, the *touched-atomic* bit of a word is set on the first read of the word in a critical section or for a word prefetched from L2/memory or from a remote L1 in *touched-atomic* or *Registered* state. The bit is reset on an acquire or a lock hand-off, including the end of the phase. A read to *Valid* data with *touched-atomic* bit set or with an address that misses in the access signature is considered a hit. Otherwise, the *Valid* data is no longer up-to-date and must be marked invalid and a read miss request is issued.

In Figure 3.2, assume that variables *a* and *b* are in the same cache line. Then *C1*'s `load b` will be a hit since *C1*'s `load a` will bring in *b* as well and set its *touched-atomic* bit. On the other hand, `load b` in *C2*'s second critical section is a miss. This is because the preceding `load a` will read *a* in its own cache in *Registered* state and so will not prefetch *b* which is registered at *C1*.

Finally, we note that using a single, plain Bloom filter at each core to determine what to invalidate is inherently conservative. For example, it is possible that an address may have been updated before it had been last seen by a core but not updated again since then; our system will still invalidate the address on a read (in the same phase) from that core. In addition, false positives in a finite Bloom

	Lock request from core *i*	Unlock request from core *i*	Response for lock request from core *i*	Remote lock request from core *k*
LockQ	set *Locked*	**if** *nextPtr* != null send response to *nextPtr*; go to *Invalid* **else** unset *Locked*	unstall core *i*; merge received signature	**if** *Locked* is set *nextPtr* := *k* **else** send response to core *k*; go to *Invalid*

Invalid	stall core *i*;
	update tag;
	go to *LockQ*;
	set *Locked*;
	send lock request to L2
	(writeback if needed)
		send unlock request to L2	X	**if** sig-only request
				send response to core *k*
				else
				send *Nack* to L2

(a) L1 cache for core *i*

	Lock request from core *i*	Unlock request from core *i*	Lock/Unlock/WB/Nack response from memory for core *i*	Lock writeback from core *i*	Nack from core *i* for core *k*

Valid	**if** *WB* == 0
	fwd req to *tailPtr*;
	else // *WB* == 1
	if *Locked* is not set
	send sig-only req to *lastAcquirer* for *i*;
	WB := 0
	else // *Locked* is set
	if *firstWaiter* != null
	fwd req to *tailPtr*
	else
	firstWaiter := *i*;
	tailPtr := *i*
		if *firstWaiter* != null
		send sig-only req to *i* for *firstWaiter*;
		WB := 0
		else
		unset *Locked*
			X	**if** *firstWaiter* == null
				copy *Locked* from WB message;
				lastAcquirer := *i*;
				firstWaiter := *nextPtr*;
				WB := 1;
				else // race
				if *Locked* is not set
				send sig-only req to *i* for *firstWaiter*
					if *WB* == 0
					firstWaiter := *k*
					else
					if *Locked* is not set
					send sig-only req to *lastAcquirer* for *k*;
					lastAcquirer := null
					else
					firstWaiter := *k*

Invalid	update tag;
	send data req to memory;
	(writeback if needed)
		update tag;
		send data req to memory;
		(writeback if needed)
			if not tag match
			allocate line;
			update tag;
			(writeback if needed)
			go to *Valid*;
			apply actions for Lock/Unlock/WB/Nack as specified in *Valid*
				update tag;
				send data req to memory;
				(writeback if needed)
					update tag;
					send data req to memory;
					(writeback if needed)

(b) L2 cache

Table 3.1: State transitions for a lock word. *X* indicates unreachable states.

filter cause valid addresses to be invalidated if the filter entry is updated by another address mapped to the same entry. Another source of imprecision occurs when the signature is transferred well after the lock release occurs. Such a signature may include addresses to accesses after the release and before the subsequent acquire – these do not precede the acquire by happens-before and may lead to false positives and unnecessary invalidations. Our evaluation, however, showed that such cases did not occur often for applications with reasonable lock synchronization; nevertheless, we later discuss some approaches to mitigate such effects (Section 3.7).

End of phase actions: At the end of a phase, as with DeNovo, we insert self-invalidation instructions for all regions with writable effects in that phase. This includes atomic and non-atomic regions. Analogous to DeNovo, all data in such regions is invalidated unless it is registered or its *touched* bit (for non-atomic regions) is set or its *touched-atomic* bit (for atomic regions) is set. All *touched* and *touched-atomic* bits are reset at the end of the phase and all Bloom filters are cleared.

3.4.2 Lock Implementation

Tables 3.1a and 3.1b describe the state transitions for the L1 and L2 caches respectively for lock words, building on top of the DeNovo line protocol (as with DeNovo, the coherence states are at word granularity). We next discuss these in detail.

L1 transitions: There are two states at L1 for a lock word: *LockQ* and *Invalid*. The lock word transitions to *LockQ* on receiving a lock request from its core, and stays there until it transfers the lock (along with the access signature) to *nextPtr* or until the line is evicted. While in *LockQ* state, a bit in the data portion of the lock entry, called *Locked*, indicates whether the lock is held or released. Figure 3.3 shows the lock word layout at the L1 with a lock queue.

On a lock request by a core, its L1 sets the *Locked* bit for the corresponding word. If the word was already in *LockQ* state, the L1 informs the core of a successful lock acquire. If the previous state was *Invalid*, a lock request is sent to the L2 and the core is stalled (the cache does not service any further requests from the core) until the response is received.

On an unlock request to *LockQ* state, if *nextPtr* is not null, the L1 transfers the lock to the *nextPtr* core and transitions to *Invalid*. Otherwise, it unsets *Locked*. An unlock request to *Invalid* state generates a request to the L2. This request is simply a notification and does not bring back the cache line (the state stays *Invalid*).

An L1 in *LockQ* state may receive a remote lock request forwarded by the L2. If the *Locked* bit is set, the request is queued in *nextPtr*; otherwise, it is serviced immediately by transferring the lock and changing the state to *Invalid*. The L1 may also receive a remote lock request in *Invalid* state due to a previous writeback. If this request is only for the signature, it transfers the signature (along with an implicit lock transfer) to the remote requestor. If the request is for the lock as well, then it signifies a race between the L1's writeback and the remote request at the L2. In this case, L1 returns a Nack to the L2 – we discuss how the L2 responds to the Nack in detail below.

Eviction of lines with lock words at the L1 is similar to DeNovo's L1 evictions (not shown in Table 3.1a). The main difference is that the writeback message needs to indicate which words are in *LockQ* state so that the L2 can perform appropriate action as discussed below. Table 3.1a does not show any action for writeback requests generated by L2 for L1. This is because the L2 does not need to maintain inclusion with the L1 for lock words (similar to *Valid* data in DeNovo). The distributed lock queue constructed in the L1s stays valid and does not need to be rebuilt on an L2 writeback.

L2 transitions without L1 writebacks: The L2 has two states – *Invalid* and *Valid*. The main source of complexity at the L2 comes from L1 writebacks of *LockQ* words; we therefore first discuss L2 transitions without L1 writebacks, indicated by *WB*=0 in Table 3.1.

On a lock request in *Valid* state, the L2 forwards the request to its *tailPtr* core and updates the *tailPtr* with the requesting core's ID. A lock request in *Invalid* state allocates the line for the lock word, triggers a fetch from memory, and keeps the L2 in *Invalid* state. When the response returns, the L2 transitions to *Valid* and applies the actions for the *Valid* state to the lock request (i.e., forwards the request to *tailPtr*). If the line was deallocated between the request and the response due to eviction, another line is allocated and the above action taken.

An unlock request in *Valid* state can only occur if the unlocking L1 previously performed a writeback on the lock (i.e., *WB=1*), and so is discussed below.

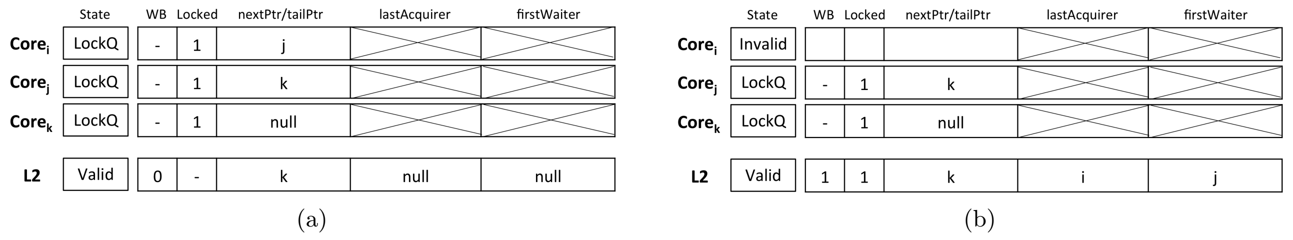
Writebacks generated by the L2 to memory are similar to DeNovo. As we see below, all the lock queue related information needed at the L2 is maintained as part of the lock word in the L2 – on an L2 writeback, this information is simply preserved at memory and made available to the L2 for later use.

Handling L1 lock writeback at the L2: When the L2 receives a writeback from an L1, it must ensure that it stores all information needed to construct the lock queue that was stored at the L1. This information is stored in the data portion of the L2 along with the *tailPtr*. An L1 writeback containing a lock word can originate only from the head of the lock queue in *LockQ* state because other cores are either stalled on their lock request or invalidated after transferring the lock. The L2, therefore, stores the following information in its data portion on an L1 writeback from core *i* (Figure 3.3 illustrates the L2 data layout with example values before and after the writeback):[1]

WB: The *WB* bit is set to 1 to indicate that the lock has been evicted from the L1 of the head of the lock queue.

Locked: The *Locked* bit from the writeback message is copied into the L2 to indicate whether the lock was released (*Locked*=0) at the time of the writeback.

[1]Storing these fields in the data bank of the L2 does not limit the number of cores that can be supported as we can increase the data size of a lock variable as needed.



(a)

	State	WB	Locked	nextPtr/tailPtr	lastAcquirer	firstWaiter
Core$_i$	LockQ	-	1	j	☒	☒
Core$_j$	LockQ	-	1	k	☒	☒
Core$_k$	LockQ	-	1	null	☒	☒
L2	Valid	0	-	k	null	null

(b)

	State	WB	Locked	nextPtr/tailPtr	lastAcquirer	firstWaiter
Core$_i$	Invalid				☒	☒
Core$_j$	LockQ	-	1	k	☒	☒
Core$_k$	LockQ	-	1	null	☒	☒
L2	Valid	1	1	k	i	j

Figure 3.3: Example showing L1 and L2 data layout for the distributed queue-based lock (a) before writeback and (b) after writeback.

lastAcquirer: L2 sets *lastAcquirer* as *i*. This is used to forward the next lock requestor to core *i* to obtain the access signature.

firstWaiter: L2 copies *nextPtr* from the writeback message into its *firstWaiter* field to indicate the first element in the queue after the head. On a subsequent unlock, the lock must be transferred to the *firstWaiter* core if it is not null.

Next we revisit the transitions for various messages at the L2 when the *Valid* state has *WB*=1. On a lock request, if *Locked* is not set (writeback occurred after lock release), L2 forwards the request to the *lastAcquirer* core. This request is for the access signature only since we already know that the lock has been released. If *Locked* is set (writeback before release), then L2 checks if *firstWaiter* is null. If it is not null, then L2 queues the request by forwarding it to *tailPtr*. Otherwise, it sets *firstWaiter* to *i* since there is no other waiter in the queue.

Similarly for unlock requests, if *firstWaiter* is not null, L2 forwards the request of *firstWaiter* to *lastAcquirer* for the signature (and implicit lock transfer). Otherwise, the queue is empty. L2 resets *Locked*, indicating that the evicted head is unlocked now and is ready to transfer the lock.

Handling races: There can be a race between an L1 lock writeback from core *i* and a request for the same lock from another core *k*. Thus, before getting the writeback, the L2 can forward core *k*'s request to L1. In this case, L1 nacks the request back to L2, which takes the following actions depending on whether it has already received the writeback (last column of Table 3.1b):

The Nack arrives before the writeback (WB=0): L2 simply sets *firstWaiter* to core *k*. When the writeback arrives, L2 finds its *firstWaiter* is not null and its request must be handled. If the *Locked* bit in the writeback is unset, L2 knows the lock was released and so can forward *firstWaiter*'s request to core *i* for signature transfer. If the *Locked* bit is set, then nothing needs to be done; the lock transfer to core *k* will occur when the *Unlock* arrives.

The Nack arrives after the writeback (WB=1): L2 services core *k*'s request using the information stored in the writeback; if *Locked* is not set, the request is forwarded to *lastAcquirer*. Otherwise, *k* is stored as the *firstWaiter*.

The above race is the only one that occurs in the lock protocol. It involves at most two cores and results in exactly one possible Nack message that the L2 immediately handles, with no deadlock or livelock causing actions.

3.4.3 Exception Handling

DeNovoND requires DeNovoD's exception handling mechanisms (as described in Section 2.5) to be extended to consider hardware Bloom filters. Bloom filters are irreproducible metadata about memory updates, so they must be properly preserved across exceptions. Adding support for Bloom filters in the following scenarios may increase their overheads, but we project that the overheads will be amortized over many cycles without exceptions.

Context switches: If a Bloom filter is not empty and is being used for update tracking at a context switch, the contents of the filter should be saved along with other information. As for possible interfering invalidations from other contexts, on-the-spot self-invalidations used for accesses in atomic regions never invalidate other processes' valid data because invalidations are performed only on an actual read.

Thread migration: The Bloom filter on the core from which a thread is migrated should be moved to a new core and reset. If the filter is highly saturated, we can set the filter at the migrated core to all 1's to conservatively invalidate on a read. This can allows us to save the traffic overhead to transfer the filter.

Error recovery: In the common case where read-only copies are not included in a snapshot and recovered execution always misses on such data, hardware Bloom filters do not need to be saved and restored. If read-only copies are included in a snapshot and restored after recovery, hardware Bloom filters should also be preserved to trigger invalidations for stale data.

3.5 Overheads

DeNovoND incurs the following overheads over DeNovo.

Hardware Bloom filter: There is one Bloom filter per core. A conservative upper bound for its size is the virtual memory size. In practice, an effective size can be empirically determined by measuring the number of atomic writes to distinct addresses in various applications. The size must also be large enough to have tolerable false positive rates. In our system, a relatively small size Bloom filter of only 256 bits worked well and provided performance similar to an infinite size Bloom filter for most cases. This is because the size of the key domain is restricted only to the addresses in atomic regions, and the filter is flash cleared at the end of a phase.

The quality of the hash function also impacts the efficiency of Bloom filters [109]. We experimented with two hash functions, multi-bit selection (similar to the one used in [31]) and H_3 (universal hash function that provides uniformly distributed hash values [29]), which showed consistent performance across applications. For our evaluation, we used H_3 which worked better with applications with high false positive rates. Finally, [31] has shown that Bloom filter operations of element insertion, membership query, and flash clear can be implemented very efficiently in hardware.

Storage overhead: Our distributed queue-based lock protocol reuses the L1 and L2 cache data banks to store the waiter queue information, incurring zero storage overhead for that purpose. It requires one additional state *LockQ* at L1 to distinguish between lock and data words. This does not result in any added storage overhead for L1 state as DeNovoD already requires two bits per word for storing three states (*Invalid, Valid, and Registered*). With an additional *LockQ* state, we now have four states stored in two bits. The two L2 states for lock words can reuse the L2 per-word state bit of the baseline DeNovoD protocol – lock words simply add new transitions to the existing L2 states, triggered by lock related messages. Thus, the lock protocol does not incur any additional storage overhead. The externally visible protocol states for data accesses also stay the same as for DeNovo. For efficient tracking of atomic writes, however, we added a *touched-atomic* bit per word in the L1 as an additional state bit (used only by the local core).

Communication and computation overhead: On acquire/release, the Bloom filter of the releaser is piggybacked on the lock transfer message. In order to minimize impact on network traffic, we can

Processor Parameters	
Core frequency	2GHz
Number of cores	16
Memory Hierarchy and Network Parameters	
L1 data cache	64KB, 64 bytes (16 words) line size
L2 (16 banks, NUCA)	16MB, 64 bytes line
Memory	4GB, 4 on-chip controllers
L1 hit latency	1 cycle
L2 hit latency	29 to 61 cycles (bank-dependent)
Remote L1 hit latency	35 to 83 cycles
Memory hit latency	197 to 261 cycles
Network parameters	2D mesh, 16 bit flits
Bloom filter size	256 bits (infinite for reference)
hash function	4 H_3

Table 3.2: Simulated system parameters for DeNovoND.

compress the Bloom filter using run-length encoding as in [31] or a Bloom-filter specific compression technique [95]. In our evaluations, we conservatively do not model such compression and charge the full 256 bits (32 bytes) of network traffic for the Bloom filter at a lock transfer. When a core receives a lock transfer message along with the signature, it needs to merge the received Bloom filter with its own before executing memory instructions in the critical section. The time for merging can be partially hidden by not blocking the execution until the first write/read instruction to an atomic region is issued.

For the distributed queue-based lock, there is an additional overhead for writeback messages which need to include an additional bit per word to indicate if the word is in *LockQ* state so that the L2 can perform appropriate lock related actions for this word. This overhead, however, can be compensated by observing that the writeback message does not have to contain full lock words, but only the *Locked* and *nextPtr* parts. The queue-based lock protocol also requires new state transitions in response to lock related messages; however, these do not introduce any new transient states or interact with the data protocol and can be separately verified.

3.6 Methodology

For our simulations, we use the same Simics [83] full-system functional simulator with the Wisconsin GEMS memory timing simulator [88] and the Princeton Garnet [10] interconnection network simulator as used in Chapter 2. Table 3.2 shows the key parameters of our simulated systems. We use the Bloom filter implementation shipped with GEMS [88] with the H_3 hashing function and 256 single-bit entries.

We also simulated configurations with infinite Bloom filter entries for reference. For the signature transfer, we add a 256 bit (32 byte) payload to the lock transfer message and simulate network traffic and latency accordingly. This is a conservative estimate since the signature could be compressed.

3.6.1 Simulated Systems

Our distributed queue-based lock is specifically designed for DeNovoND, reusing the coherence states of DeNovo, with no added transient states and limited race interactions. Implementing it on a conventional MESI-like protocol is possible, but will involve far more complexity to deal with interactions with the already existing numerous transient states and race conditions. On the other hand, comparing DeNovoND with distributed queue-based locks and MESI with conventional locking may not be fair to MESI. We therefore implemented simplified (idealized) queue-based locks that work for both MESI and DeNovo to isolate the effectiveness of access signatures. This idealized implementation maintains a "lock table" which is keyed by a lock variable address and maintains the waiter queue for each lock. Accesses to this table – creating an entry and grabbing the lock, adding a core to the waiter queue, waking up the first waiter in the queue, etc. – do not incur extra cycles. We also do not charge traffic overhead for lock and signature transfer for the idealized lock. Once a core is ready to release the idealized lock, lock transfer is instant and the next requestor wakes up immediately. Hence we evaluated the following systems:

MESI: We simulated MESI using idealized queue-based locks (MIL) and the `POSIX pthreads` mutex library (MPL). We modified the original implementation of MESI in GEMS [88] to support non-blocking writes for a fair comparison with DeNovoND where writes are non-blocking by default. Atomic instructions used in `pthreads` mutex codes are simulated using blocking store fences for correct execution.

DeNovoND: We simulated DeNovoND with idealized queue-based locks (DIL) and with distributed queue-based locks (DQL), both with a 256 bit Bloom filter (DIL-256 and DQL-256)) and, for reference, an infinite size Bloom filter (DIL-inf and DQL-inf). For DQL, operations on the lock incur latency consistent with table 2.2. For the signature transfer, we add a 256 bit (32 byte) payload to the lock transfer message and simulate network traffic and latency accordingly. This is conservative for DQL-256 since the signature could be compressed. It is aggressive but reasonable for DQL-inf since DQL-inf is intended to be a best case reference model.

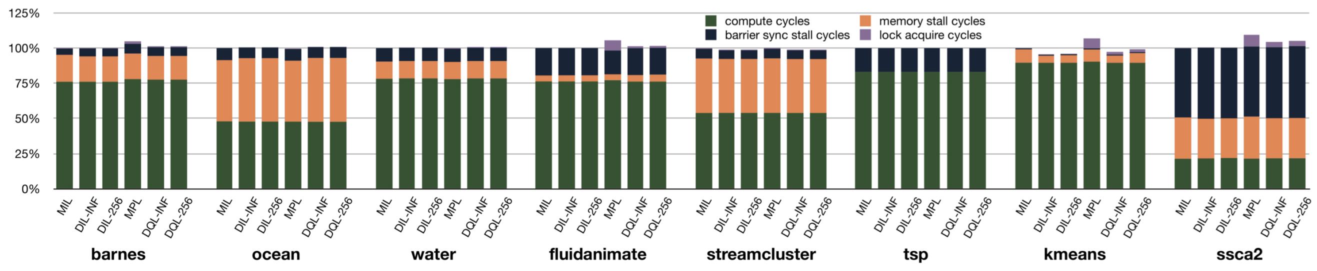
3.6.2 Workloads

We evaluated 11 benchmarks with lock synchronization, taken from various suites to represent a range of behavior such as lock frequency, lock granularity, contention, critical section length, and shared working-set size. We evaluated *barnes* (16K particles), *ocean* (258×258), and *water* (512 molecules) from SPLASH-2 [123]; *fluidanimate* (35K particles) and *streamcluster* (8,192 points) from PARSEC 2.1 [19]; *tsp* (17 cities) as used in [24]; and *kmeans* (8,192 points, 24 dimensions, 16 centers), *ssca2* (2^{13} nodes), *genome* (256 nucleotides), *intruder* (1,024 traffic flows), and *vacation* (16,384 records) from STAMP [94].

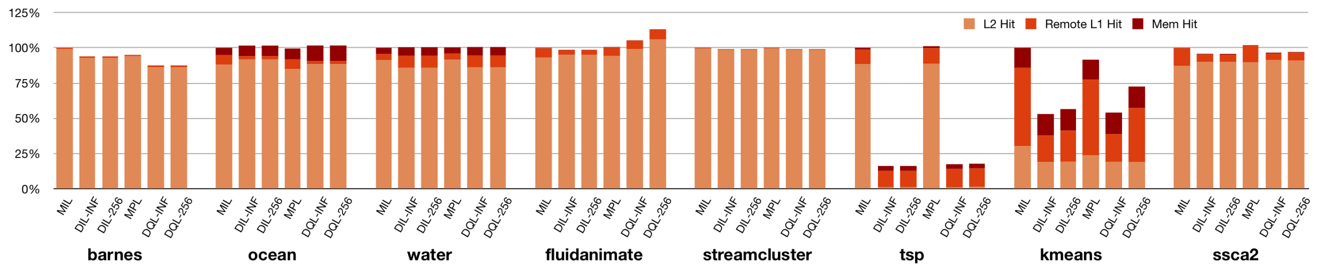
The benchmarks from SPLASH-2 and PARSEC represent traditional applications designed and optimized to scale well with lock synchronization. The benchmarks from STAMP and *tsp*, however, were originally designed for hardware and software transactional memory. We ported them to use locks for our simulated systems. For short transactions, we directly replaced them with critical sections (*tsp*, *kmeans*, *ssca2*, and *intruder*). For longer transactions, we used finer-grained locks (*genome*, *vacation*).

We found that 3 out of the 6 transactional applications (*genome*, *intruder*, and *vacation*) spent $> 70\%$ of their execution time on lock acquire for all studied configurations. Clearly, parallelization using lock synchronization is inappropriate for these applications, for both MESI and DeNovoND. We therefore focus our results on the other 8 applications, referring to them as "lock-efficient" applications (Section 3.7.1). For completeness, we separately report results for the above three lock-inefficient applications (Section 3.7.2). We discuss optimizations to improve the performance of DeNovoND for the lock-inefficient applications, but fundamentally, these must be parallelized using different techniques for reasonable parallel speedups. Such techniques (including possibly transactional memory) are outside the scope of this work.

Finally, the lock-inefficient applications showed significant non-determinism in execution time. Although our timing simulations are deterministic, they depend on the state of the system when the application is started (the Simics checkpoint at the start of the application). For different state, the lock-inefficient applications showed varying results. We therefore ran each such application with five different checkpoints for each system and averaged the results (the same five checkpoints are used for all systems). We also report the results for the lock-efficient applications averaged across three different



(a) Execution time.



(b) Memory stall time.

Figure 3.4: Execution time and memory stall time of lock-efficient applications on 6 configurations, normalized to MIL.

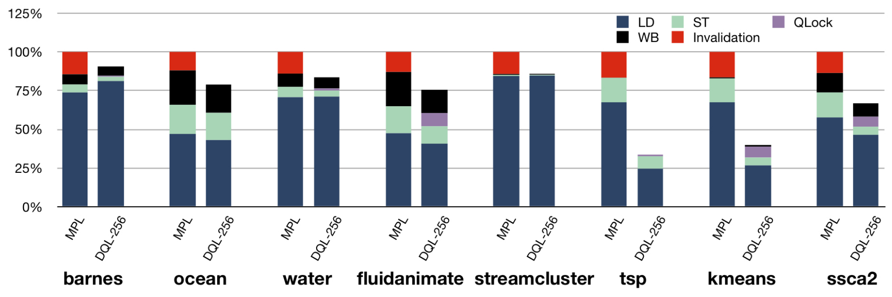
checkpoints, but these applications did not show much variability across their checkpoints.

We manually analyzed and annotated all benchmarks as we would do for DPJ programs. We used the same simulator API as in Chapter 2 to convey region information to the hardware and perform self-invalidations.

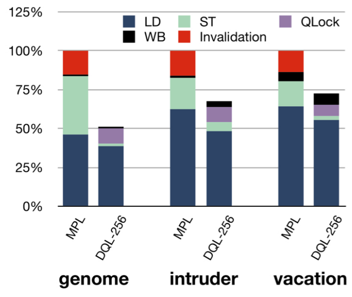
3.7 Evaluation Results

3.7.1 Lock-Efficient Applications

Figure 3.4a shows the execution time for our 8 lock-efficient applications for the 6 configurations described in Section 3.6.1. All bars are normalized to MIL. Each bar is divided into compute time, stall time due to data memory accesses (henceforth referred to as *memory time*), barrier time, and lock acquire time. Since we model non-blocking lock releases, lock release time is negligible. Since our focus is on the memory system, Figure 3.4b blows up the memory time in each bar of Figure 3.4a, divided into stalls for L1 misses resolved at L2, a remote L1, or main memory. Since all modeled systems implement non-blocking stores, virtually all memory stalls are due to loads. Figure 3.5a presents network traffic



(a) Network traffic (lock-efficient).



(b) Network traffic (lock-inefficient).

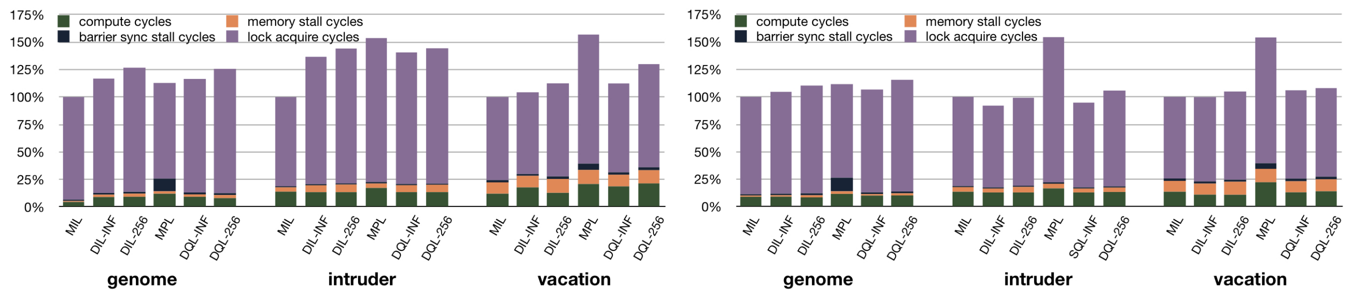
Figure 3.5: Network traffic of all applications on MPL and DQL-256, normalized to MPL.

for the same applications on MPL and DQL-256 (normalized to MPL), classified by the message type: load, store, queue lock/unlock, writeback, and invalidation. The queue lock/unlock traffic exists only in DQL-256 for transferring distributed queue-based locks with signatures. For MPL, the lock traffic is aggregated with the data load and store traffic. Note that only MPL incurs invalidation traffic. We do not show network numbers with other configurations because they are idealized, but we confirmed that the network results for DQL-256 stay qualitatively similar even when compared to MIL.

MIL vs. DIL-inf: For all 8 applications, DeNovoND shows the same or slightly better (up to 5%) execution time compared to MESI with idealized locks and infinite length Bloom filter. Focusing on memory time, again DIL-inf is either the same or better than MIL. For some applications, DIL is much better than MIL; e.g., 47% and 84% better for *kmeans* and *tsp* respectively. This is because MIL suffers from false sharing while DIL does not due to its per-word coherence state.

MPL vs. DQL-inf: Comparing the realistic lock implementations (but still with infinite Bloom filter size), we find that for all 8 applications, DQL-inf shows comparable or slightly better execution time than MPL. In fact, even compared to the idealized lock implementation in MIL, the execution time for DQL-inf is about the same or better in 7 of 8 cases and only 4% worse in the remaining case (*ssca2*). In terms of memory time, again DQL-inf is either comparable or sees large benefits due to the lack of false sharing relative to both MPL and MIL.

Impact of finite signatures: We next evaluate the impact of restricting the Bloom filter size: DIL-inf vs. DIL-256 and DQL-inf vs. DQL-256. The 256 bit Bloom filters show virtually the same execution times as the infinite length filters. In terms of memory time, the two Bloom filter sizes are similar for 6 of the 8 applications. For *fluidanimate* and *kmeans*, however, the 256 bit filter shows a degradation.



(a) Baseline. (b) "Write-Once" Atomic Region and Signature Clearing.

Figure 3.6: Total execution time of lock-inefficient applications on six configurations: (a) baseline, (b) with "write-once" atomic region optimization and signature clearing (threshold=99%) applied, normalized to the MESI with idealized locks (MIL) configuration.

For *kmeans*, memory time for DQL-256 continues to remain significantly better than for both MESI configurations (20% or more better), but for *fluidanimate*, it is worse by 13% (the only application where this is the case).

Fluidanimate and *kmeans* show the above behavior due to a confluence of a few subtle effects. First, both use critical sections where an atomic region address that is read is also written. Often an atomic region address read by a core was also last written by the same core (either in the previous phase or in a previous critical section). If this address is still in the core's cache in modified (for MESI) or registered (for DeNovoND) state, then the read will be a hit for both MESI and DeNovoND. Otherwise, if the address was written back, the read will be a miss for both MESI and DeNovoND. The difference between the protocols arises for any other atomic region addresses that come along with such a read miss as part of the same cache line. If the same core reads such an address in a subsequent critical section without an intervening write by another core, then MESI will still hit in the cache but DeNovoND will have to check against the Bloom filter. This could require a self-invalidation since the corresponding Bloom filter bit may be set, resulting in an extra miss over MESI. A smaller Bloom filter exacerbates this problem since it also results in false positives on the key domain. Further, the effect is more noticeable in DQL than in DIL because *fluidanimate* and *kmeans* have fine-grained locks – these locks pollute the cache and cause more replacements, exacerbating the above effect.

Network traffic: Figure 3.5a shows that for all the applications, DQL-256 has much lower traffic than MPL (33% on average, 67% maximum). This directly translates into energy reduction.

The primary sources of these savings in DeNovoND are as follows: (1) DeNovoND does not incur any traffic for invalidations, a significant effect in all applications. (2) Store traffic is reduced in some applications because store requests in DeNovoND do not bring in the cache line – they directly write into the L1 word and only send out a registration request for that word (multiple registrations for a given line are combined and sent on the network as mentioned in Chapter 2). (3) The net reduction in load misses (memory time) due to the lack of false sharing (Figure 3.4b) directly leads to lower load traffic in several applications. (4) Load traffic is further reduced because a load response only contains valid or registered words of a cache line. Since coherence state is preserved per word, some words may be invalid at the servicing cache.

A source for increased network traffic in DeNovoND is the 32 byte signature with all lock transfers. Figure 3.5a shows that this is small in all our applications. It can be further reduced through compression techniques.

Summary: Overall, our results show that for these applications, the access signature mechanism allows DeNovoND to enjoy all the benefits of DeNovoD even in the presence of lock-based synchronization. Further, the signature size needed is small (32 bytes).

3.7.2 Lock-Inefficient Applications

The lock-inefficient applications spend more than 70% of their time on lock acquires, but are presented here for completeness. Figure 3.6a shows their execution times analogous to Figure 3.4a. There are several ways in which these applications differ from the lock-efficient ones. First, as mentioned earlier, they are dominated by lock acquire time and so need a significantly different algorithm for parallelization and/or synchronization. These applications were originally designed to study transactional memory. Some of them use patterns for which lock-free synchronization is commonly used.

Second, as discussed in Section 3.6.2, these applications show significant non-determinism. Although we report results averaged over five runs starting from five different Simics checkpoints (the same five checkpoints for each system), the variability makes comparing different systems difficult.

Third, we find that compute time varies across different systems for each of these applications. Although not shown here, a significant fraction of compute time comes from the OS (e.g., due to frequent memory allocations), forming the main source of the compute time variation. (The lock-efficient applications have negligible OS compute time.) Our results must therefore be understood in the context of the above caveats.

MIL vs. DIL-inf: For all three applications, DIL-inf shows observably worse performance than MIL (16% for *genome*, 36% for *intruder*, and 5% for *vacation*). A large part of the performance difference appears to come from acquire time; e.g., DIL-inf spends 40% more cycles waiting for lock acquisition than MIL with *intruder*. Though memory time is a very small portion, it affects acquire time by increasing the time spent within critical sections. Our detailed results show that DIL-inf suffers from higher memory time than MIL, especially for *genome* and *intruder*.

The higher memory time above occurs due to an access pattern where an address is written only once in a phase and then read several times. Specifically, *genome* and *intruder* use list and hash table data structures that store "data" or "key-data" pairs of each entry as a field of the entry object – in these programs, the data is initialized when a new element is inserted (within a critical section) but never modified afterwards. A core may read this data later in different critical sections – DeNovoND will self-invalidate on such reads since it does not know if there was an intervening write since the last read. MESI, on the other hand, will hit on such reads if they happen close enough to exploit temporal locality.

Section 3.7.3 discusses how we can use software information to remedy the above situation. We believe, however, that a better solution to this problem is a better synchronization construct – using locks for such reads is overkill. Such constructs in the context of DeNovoD and DeNovoND are a key part of our future work.

MPL vs. DQL-inf: DQL-inf performs slightly worse than MPL with *genome* for the same reason as the comparison between MIL and DIL-inf. DQL-inf outperforms MPL with *intruder* and *vacation* – for these applications, MPL has significantly higher acquire time than MIL. MPL's pthread locks, however, are inherently inefficient with high lock contention; therefore, this is not a fair comparison for MESI. Thus, little can be deduced here except perhaps that DeNovoND performance seems to be in the same range as MESI (this inability to draw a conclusion is an inherent artifact of the problem studied).

Impact of finite signatures: With smaller Bloom filter sizes, false positives exacerbate the impact of the conservative invalidations described above; for *genome* and *intruder* – DIL-256 and DQL-256 perform worse than DIL-inf and DQL-inf by 4% to 10%.

Vacation does not suffer from the conservative invalidations of *genome* and *intruder*, but reveals a different source of inefficiency with smaller signatures. Figure 3.6a shows DIL-256 is 8% worse than DIL-inf, while DQL-256 is 17% worse than DQL-inf for this application. This is mainly due to its large working set of atomic data, which can increase the false positive rate if a Bloom filter is too small. In addition, *vacation* has only one phase without any barriers in between; thus the Bloom filters get filled up for a long period without clearing. This further exacerbates the false positive rate, resulting in unnecessary self-invalidations and higher memory times. Section 3.7.3 describes an optimization technique called signature clearing to deal with this issue.

Network traffic: Figure 3.5b shows network traffic of the lock-inefficient applications on MPL and DQL-256. DQL-256 generates less network traffic (up to 48%) than MPL for all three applications for reasons similar to that for the lock-efficient applications. In addition, with relatively high lock contention, repeated accesses to lock variables can generate increasingly higher network traffic in MPL. In contrast, distributed queue-based lock request/response traffic scales in proportion to the number of lock transfers.

3.7.3 Optimizations

Handling "write-once" atomic data: As with the case with *intruder* and *genome*, once a new entry is created and then inserted into a data structure (list, hash table, etc.), the "data" portion of the entry may remain read-only for the entire execution while other fields of the entry are modified as the structure grows or shrinks. In this case, classifying the "data" as atomic makes every self-invalidation after the very first one (the memory location may have been used and freed before) unnecessary.

DeNovoND can safely get rid of these invalidations by identifying such atomic accesses as made to a "write-once" atomic region. In addition to general information about atomic regions and effects, software can allow such "write-once" atomic data to be marked differently by using a special region ID or a special op-code for the write. Then DeNovoND can exploit it to prevent such data from being self-invalidated as follows. If the data is known to be in a "write-once" atomic region, DeNovoND does not reset its *touched-atomic* bit on lock transfer; therefore, when the data is accessed (read) again later, it is treated as if it has been already accessed in the same critical section (with *touched-atomic* bit set) and will not be self-invalidated, thereby eliminating several subsequent misses.

The write-once annotation can be considered to be a generalization of *final* variables in Java; a final variable can only be initialized once, either at the time of declaration or by the constructor of the class in which it is declared [100]. Our write-once variables must be written (at most) once per parallel phase.

Signature clearing: Depending on the atomic write-set size in a phase, the fixed-size hardware Bloom filter may get saturated (all bits set) before the phase is over. This drives the false positive rate very high, resulting in many unnecessary self-invalidations. Saturated Bloom filters can be flash-cleared by a simple hardware operation, but it also requires flushing out atomic words in the cache. Also, the fact that a signature has been cleared in the releaser should be propagated to the acquirer so that the acquirer can update its cache according to the new version of the Bloom filter. We implemented a signature clearing algorithm that carries a vector of clearing counters per core. When signature clearing is triggered on a core, its counter is incremented. The vector of clearing counters is transferred on a lock transfer along with the access signature. The acquirer compares the received vector with its own, and performs signature clearing if there exists an element in the received vector that has a larger counter than the corresponding element in its own vector. Before the lock is transferred again, the vector is updated to have up-to-date values.

Performance impact: Figure 3.6b presents execution times analogous to figure 3.6a, but with the above optimizations applied.

For *genome*, all DeNovoND protocols now perform comparable to the MESI counterpart. Our detailed results show large reductions in memory time from the write-once optimization (118% to 151%). Since this reduction mainly comes from atomic accesses within critical sections, lock contention also improved. *Intruder* shows similarly dramatic results in memory time improvement with consequently large improvements in execution time for the DeNovoND configurations; acquire time is reduced by 36 to 42%, memory time by 56 to 76%, and overall execution time by 43% on average.

For *vacation*, DIL-256 and DQL-256 (protocols with finite Bloom filters) show performance benefits from signature clearing; DIL-256 and DQL-256 were 17% and 8% worse than DIL-inf and DQL-inf respectively without signature clearing. With signature clearing, with 99% filter saturation percentage as the trigger for clearing, the difference is reduced to 5% and 2%.

Overall, the optimizations are quite effective, making the DeNovoND protocols comparable or better than the corresponding MESI protocols even for the lock-inefficient applications.

3.8 Summary

This chapter takes a significant step towards a vision for complexity-, performance-, and energy-efficient multicores enabled by disciplined shared-memory programming practices. Chapter 2 showed how this vision could be achieved for deterministic programs. This chapter develops DeNovoND, a system that additionally supports disciplined non-determinism with minimal additional overheads and complexity relative to DeNovo.

DeNovoND exploits a previously developed software-level guarantee that non-deterministic (atomic) data accesses are distinguishable and protected by a lock. The key insight is to use small and simple hardware Bloom filters to track and communicate such accesses across lock transfers, preserving DeNovo's previous advantages of no transient states, directory overhead, invalidation messages, or false sharing. Underlying the data transfer mechanism is a distributed queue-based lock mechanism that uses the cache data banks to construct a lock-waiter queue, without additional state bits or directory storage.

DeNovoND provides comparable or better performance than MESI with the lock-efficient programs studied here. Further, network traffic is significantly reduced, impacting energy. We also identified some patterns in lock-inefficient code that did not work as well with DeNovoND – we showed optimizations to mitigate those effects, but believe the correct solution lies in alternate forms of synchronization for such codes.

Chapter 4

DeNovoSync: Supporting Arbitrary Synchronization without Writer-Initiated Invalidation

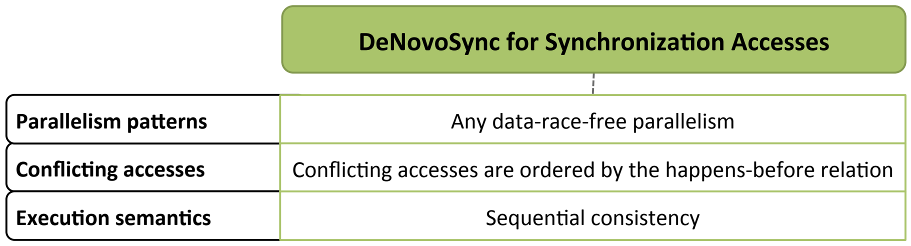
In the previous two chapters, we focused primarily on supporting data accesses, assuming restricted forms of synchronization (global barriers for determinism in DeNovoD in Chapter 2 and disciplined locks for safe non-determinism in DeNovoND in Chapter 3) with special hardware support. For DeNovo to be widely adopted, however, it must relax its software requirements and support programs with arbitrary synchronization without expensive hardware support.

In this chapter, we propose DeNovoSync, an extension to the DeNovo protocols described so far that supports programs with arbitrary synchronization without sacrificing DeNovo's benefits. Specifically, we do not add any new states to the protocol, and retain the properties of no writer-initiated invalidations and no sharer's lists. We will show that DeNovoSync can provide an efficient coherence mechanism for a variety of synchronization patterns on a coherence protocol that does not rely on writer-initiated invalidations.

4.1 Software Assumptions

As we extend DeNovo to support broader classes of applications, we relax our assumptions about disciplined software. Figure 4.1 summarizes the assumptions and execution semantics for DeNovoSync presented in this chapter. We assume that the following software properties, which are already enforced by most modern programming languages.

1. **Parallelism patterns:** Software distinguishes between synchronization and data accesses (part of the requirement for data-race-free memory models). Mainstream languages all require this; e.g., programmers are required to use volatile (Java) or atomic (C++) declarations for synchronization variables [84, 25]. We assume this distinction is conveyed to the hardware as well.



DeNovoSync for Synchronization Accesses	
Parallelism patterns	Any data-race-free parallelism
Conflicting accesses	Conflicting accesses are ordered by the happens-before relation
Execution semantics	Sequential consistency

Figure 4.1: Software assumptions and execution semantics for DeNovoSync.

2. **Conflicting accesses:** We assume that all conflicting data accesses are ordered by "happens-before" relations [6] formed by synchronization accesses. This is also part of the requirement for data-race-free.

3. **Execution semantics:** DeNovoSync provides sequentially consistent semantics for data-race-free programs.

DeNovoSync can run any applications that obey the above requirements. However, data consistency performance is improved with more information. Without further information, DeNovoSync will work correctly by invalidating all (shared, writable) data that is not registered in a core's cache at an acquire synchronization. However, with more information at acquires, these invalidations can be selective, possibly yielding better performance. This information may be in the form of compiler-specified shared writeable regions that are synchronized by the acquire (as in Chapter 2), or it may be generated dynamically through hardware signatures (as in Chapter 3). In this chapter, we assume that the program provides static regions that need to be invalidated at an acquire (details on how regions are passed to the hardware are given in Section 2.2.2). Although determining this information may be difficult in the general case,[1] for the programs we examined, it was generally easy to identify because most of them were written in a disciplined manner using high-level parallel constructs that made clear which data was being protected by which synchronization. We leave the question of how to (semi-)automatically (statically or dynamically) deduce such information in the general case to future work.

[1]To determine which region needs to be invalidated at an acquire more formally, we use the standard happens-before definition [6]. If X is the last conflicting write ordered before a read Y by happens-before, then either (1) X must be program ordered before Y, or (2) there must be an acquire A program ordered before Y such that X happens-before A and there is a self-invalidation for the region of X (which is the same as the region of Y) between A and Y. This ensures that Y never sees a "stale" value. (Note that a self-invalidation only affects non-registered data in the cache; registered data stays in the cache across synchronization boundaries.)

4.2 Design Overview

Our focus here is on correctly implementing synchronization accesses; i.e., accesses involved in a race [6]. We assume the correctness criteria for synchronization is sequential consistency, the strongest hardware-level guarantee possible. Translating this to high-level safety properties when such synchronization constructs are used in the context of otherwise disciplined code is outside the scope of this work. Without loss of generality, we assume below that there is a two level cache hierarchy with private L1 caches and a shared L2 cache.

4.2.1 The DeNovoSync0 Protocol

The following conditions are sufficient for sequential consistency of synchronization accesses [6]:

- *Write propagation:* A write is eventually visible to all cores.
- *Write atomicity:* A write is not made visible to a read until it is visible to all cores.
- *Write serialization:* Writes to the same synchronization location are serialized (i.e., seen by all cores in the same order).
- *Program order:* A synchronization access must not be issued until the previous (by program order) synchronization access completes (i.e., a write is visible to all cores and a read returns its value).

We next describe how we modify the DeNovo protocol to satisfy the above conditions.

Write propagation: Since we do not have writer-initiated invalidations, a synchronization read to a word in valid state will never see a new write. To ensure the write propagation condition, we need to perform periodic self-invalidation for such reads so that they miss and go to the last level cache and see the values of any newly registered writes. DeNovoSync0 always performs such a self-invalidation for synchronization reads to valid state; i.e., unless the word is in registered state, a synchronization read always incurs a miss.

Write atomicity: For write atomicity, DeNovoSync0 simply uses a single-reader protocol for synchronization (at the word granularity, which is the coherence granularity for DeNovo). Thus, a synchronization read is always required to register itself at the LLC and only one read can be registered at a time.

Read registration produces read-read "races" in the protocol, which we discuss further below. Since all synchronization reads seek registration, once a synchronization write is visible to a synchronization read, no later read can see an older value. This ensures write atomicity.

Write serialization: DeNovo already provides write serialization by requiring writes to be registered and allowing only a single registered write to a given location. However, with synchronization, we can have write-write races, which the baseline DeNovo protocol did not have to handle. This is discussed below.

Program order: The program order requirement is easily met by not issuing a synchronization access until the previous synchronization access (by program order) is complete (i.e., registered).

Handling races: The baseline DeNovo protocol is built on the assumption of race-freedom; i.e., there are no concurrent conflicting accesses to the same location at any time. Specifically, when a registration request reaches the LLC in registered state, the LLC immediately updates the current registrant in its registry and forwards the new request to the previous registrant – the previous registrant invalidates itself and sends an ack to the new registrant without any need to inform the directory. In contrast, many MESI protocols (including the one we study) implement a blocking transaction where the directory serves as the intermediary for transferring the ownership from the previous to the new owner – until the full transaction is completed (with all invalidation acks collected), the directory does not service any new requests to that line to avoid even more complex transient states.

We continue to maintain a non-blocking registry; i.e., the registry (LLC) forwards a new registration request to the previous registrant and continues to service other requests to the same word (including forwarding new registrations). This can result in an L1 cache receiving a forwarded registration request before it receives an ack for its own registration request by a remote L1. This is indicated by the word being in invalid state – the forwarded registration is simply stored in the MSHR entry of the pending registration. When the pending registration's ack arrives, the cache services the stored request in its MSHR, forwarding the registration ack (or data for a read) to the requesting core (keeping itself invalid). Thus, instead of serializing registrations at the LLC, we build a queue distributed among the L1 caches with pending registrations (similar to [48, 49, 117]). [2]

[2]We did not implement a non-blocking directory for MESI because of the much higher complexity. Exploring its potential impact on performance is part of future work.

```
void queue.enqueue(value v):
   node *pw := new node(v, null)
   ptr pt, pn
   loop
(1)    pt := tail
(2)    pn := pt->next
(3)    if pt == tail
(4)       if pn == null
(5)          if (CAS(&pt->next, pn, pw))
                break;
(6)       else CAS(&tail, pt, pn)
(7)CAS(&tail, pt, pw)
```

(a) enqueue().

```
value queue.dequeue():
   ptr ph, pt, pn
   loop
   ph := head
   pt := tail
   pn := ph->next
   if ph == head
      if ph.p == pt.p
         if pn.p == null return <failure>
         CAS(&tail, pt, pn)
      else
         rtn := pn->val
         if CAS(&head, ph, pn) break
   free for reuse(ph)
   return rtn
```

(b) dequeue().

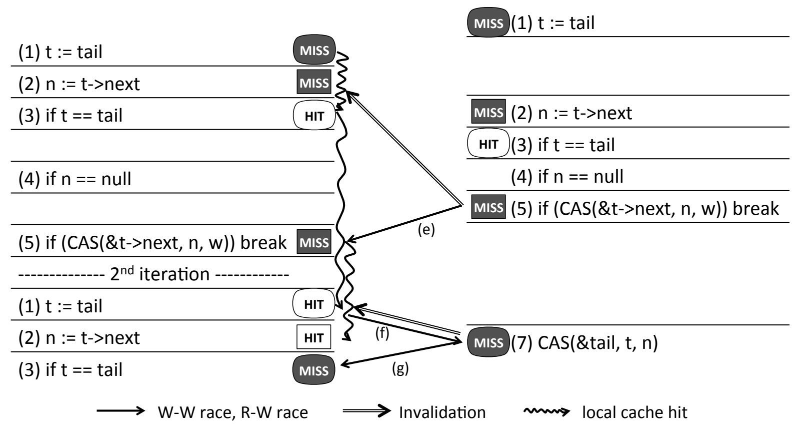
Figure 4.2: Pseudo code for Michael-Scott queue.

Summary and Example: In summary, the DeNovoSync0 protocol effectively treats a synchronization read like a read-modify-write (RMW), requiring registration. It does not add any new states to the protocol and requires only a few small changes to the actions on state transitions to accommodate registration request races.

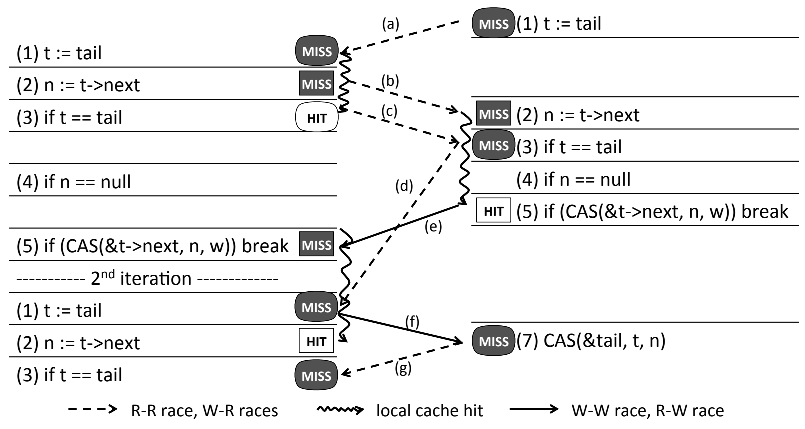
To illustrate the working of the protocol, we apply it to an example. Figure 4.2 (adapted from [110]) shows pseudo-code for the enqueue and dequeue functions for the Michael-Scott queue [91]. Figure 4.3 shows two concurrent threads executing the enqueue function with MESI (part (a)), DeNovoSync0 (part (b)), and DeNovoSync (part (c), to be discussed later) protocols. We focus on the two synchronization variables, `tail` and `tail->next` here. All solid arrows represent legitimate invalidation of stale copy or ownership transfer by R-W and W-W races respectively on MESI and DeNovo, while dashed arrows indicate invalidation caused by read registration on false races (R-R and W-R races) for DeNovo. Rippled lines connect a read hit and a read/write that brought in the value. Rounded and rectangular boxes indicate whether memory accesses to `tail` and `tail->next` in each line respectively result in a cache hit or miss.

Figures 4.3a and 4.3b show the following key differences between MESI and DeNovoSync0:

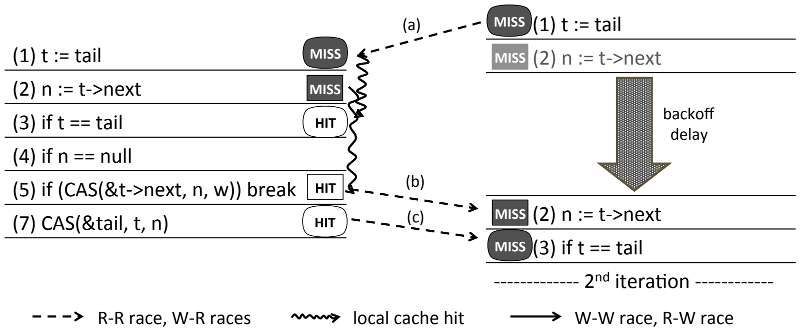
- False R-R and W-R races (edges (a), (b), (c), (d), and (g) in Figure 4.3b) trigger registration transfer and unnecessary invalidation of unchanged values for DeNovoSync0, while MESI does not incur any inter-thread communication for read accesses once valid copies are cached. These conservative invalidations on DeNovoSync0 cause the additional read misses in line 1 of the second



(a) MESI.



(b) DeNovoSync0.



(c) DeNovoSync.

Figure 4.3: Example interleavings for enqueue() of Michael-Scott queue on MESI, DeNovoSync0, and DeNovoSync.

iteration on thread 1 and line 3 on thread 2 relative to MESI.

- Rippled lines in Figure 4.3b show synchronization read/write hits enabled by earlier writes and read registrations by DeNovoSync0. Without an intervening read/write, thread 1 can safely re-

read a valid copy of `tail` in line (3) and (2) of the second iteration in Registered state (similar to MESI). In addition, on thread 2, the rippled line from line (2) to (5) enables a write hit for DeNovoSync0 as `tail->next` is brought in Registered state in line (2). MESI, on the other hand, incurs a write miss on line (5).

Overall, compared to MESI, DeNovoSync0's read registrations suffer from two extra read misses but save a future write miss. Section 4.3 qualitatively analyzes the net impact.

4.2.2 The DeNovoSync Protocol

DeNovoSync0 is a reasonable solution for certain scenarios; e.g., with a single producer and consumer. However, in many scenarios, requiring all synchronization reads to register results in unnecessary misses. Specifically, with multiple waiting consumers, the synchronization data will ping-pong between the readers unnecessarily even while there is no intervening write.

The DeNovoSync protocol continues to require registration on synchronization reads, but attempts to delay such registration requests (in hardware) based on the perceived contention in the system. It is inspired by the idea of software backoff in conventional synchronization mechanisms [9]. Hardware backoff shares with software backoff the main goal of reducing contention in the system by delaying synchronization attempts. However, hardware backoff implements more fine-grained contention control than software backoff. Efficient, finer-granularity backoff is critical for many scenarios. For example, for non-blocking synchronization (e.g., the Michael-Scott queue described earlier), there are often several synchronization reads in succession and it is not reasonable to insert a software backoff before each of them. We therefore enhance the use and scope of conventional software backoff with a hardware backoff mechanism that adaptively delays synchronization reads to a location in non-registered state as follows.

Backoff counter

DeNovoSync uses one hardware backoff counter per core to delay the core's synchronization read misses (i.e., reads to non-registered state). The size of the counter determines the maximum backoff cycles and should be determined by the system configuration; e.g., the number of cores, average miss latency, and network characteristics. The backoff counter logic performs the following operations.

Update counter

The backoff cycles value stored in the counter should adapt to the contention in the system. Our design uses *remote synchronization read requests* coming into a core as a symptom of system contention. On receiving a remote synchronization read request, a core downgrades from Registered to Valid state, sends a response to the remote requester, and increments the backoff counter. The increment is determined by a separate counter described in Section 4.2.3. The next time the core issues a synchronization read request to a value in Valid state, it stalls for the number of cycles indicated in the updated counter, delaying the issue of the synchronization read miss request. A synchronization variable in Valid state is not considered a usable valid copy. We simply reuse the state to differentiate from initial reads to Invalid state and trigger backoff delay. A synchronization read request to a value in Registered state will be a read hit and will not trigger backoff. When the counter overflows, it wraps around to zero.

Reset counter

The backoff counter is reset on a synchronization read to registered state (i.e., a read or RMW hit). Such a hit means that no other core intervened between the core's last access and the current access to the accessed location. We translate this as a low-contention status and do not delay subsequent requests.

Wakeup from backoff

When the backoff delay expires, it unstalls the core and issues the delayed synchronization read request. Since backoff delay is triggered only when the accessed location is in non-registered state, the core ignores any cached copy and issues a miss.

4.2.3 Increment Counter

The backoff counter efficiently adapts to changing contention levels in the system by increasing its value on a remote synchronization read registration request. However, a fixed increment value may not be flexible enough to handle both extremely low and extremely high contention. If the increment is fixed too low, the backoff counter may not climb fast enough to reduce unnecessary read registration requests under high contention. On the other hand, the increment of a large value may cause unnecessarily long stalls under low contention. Therefore, we introduce another level of adaptivity for the increment counter.

Update increment

Again, we use the number of incoming remote synchronization read registration requests to indicate the contention level and update the increment counter accordingly. The size of the increment and the frequency of update should be determined by considering the system configuration. We found that the number of cores is a good indicator. For example, the increment counter is increased by the default increment value on receiving every 16th remote synchronization read registration request for the 16-core system evaluated in the chapter (64th request for the 64-core system). The updated increment counter is then used to increase the backoff counter for the next remote read registration request.

Reset increment

The increment counter is reset to a default value on a release. A release indicates the successful completion of a synchronization construct and the reset prepares the core to adapt for the next synchronization.

Example

Figure 4.3c shows how the Michael-Scott enqueue() function results in a different number of misses for the DeNovoSync0 and DeNovoSync systems. On the R-R race (a), the Registered copy of `tail` at thread 2 is downgraded to Valid state and the backoff counter is updated. When the thread tries to read `tail->next` in line (2), it recognizes the backoff counter is non-zero and starts the delay (assuming `tail->next` is in Valid state).

While the read of `tail->next` is being delayed on thread 2, thread 1 can proceed with its read and write of `tail` and `tail->next`. As a result, thread 1 can escape the loop in one iteration. Depending on when thread 2 wakes up, it may or may not fail the test in line (3). Compared to DeNovoSync0 in Figure 4.3b, DeNovoSync can save three misses in thread 1 – line (5) in the first iteration and (1) and (3) in the second iteration, allowing thread 1 to finish early. As will be seen in Section 4.5, DeNovoSync quickly adapts to different levels of contention in the system. Especially under high contention, contending cores can save on numerous read and write misses that are doomed to fail and repeat.

Protocol States

We still have only three protocol states for DeNovoSync/0 (meaning "both DeNovoSync0 and DeNovoSync" from now on): *Invalid, Valid* and *Registered.* When the word is Invalid or not present in the private cache, both reads and writes request registration and bring the address in Registered state. When the word is Registered, both reads and writes hit in the cache. While DeNovoSync0 makes the same transitions for Invalid and Valid state for all synchronization requests, DeNovoSync uses Valid state to trigger backoff on a synchronization read request. To this end, DeNovoSync distinguishes synchronization read from synchronization write requests and handles them differently in the protocol – synchronization write requests will be immediately issued in Invalid/Valid states while synchronization read requests may be delayed based on the backoff counter value.

4.3 Qualitative Analysis

This section qualitatively analyzes the performance of the different synchronization mechanisms on MESI and DeNovoSync/0, using the notion of *linearization points* [57]. Commonly used synchronization objects are *linearizable*; i.e., the operations on the object appear to occur in some total order that is consistent with the program order in each thread and also with any ordering observable by threads [110]. With linearizable synchronization objects, we can typically identify a linearization point within each method whose ordering determines the ordering of the method itself. To repeat on a synchronization failure, the linearization point for acquire synchronization is usually enclosed in a loop. This loop often includes synchronization accesses that are not linearization points; i.e., accesses that do not determine if an attempt is successful. For example, the first Test part of Test-and-Test-and-Set (TATAS) or equality checks before the final CAS instruction in non-blocking algorithms do access the shared synchronization object in the same loop, but their purpose is only to filter unsuccessful attempts early before reaching the linearization point.

To systematically understand protocol overheads for each synchronization mechanism, we divide the overheads into two parts: (1) *Linearization cost* is the cost to execute the instruction at the linearization point, including all coherence activities involved. (2) *Pre-linearization cost* is the cost of executing all the other synchronization accesses in the synchronization loop. Note that the coherence overhead for data accesses protected by these synchronizations may vary between the protocols, further affecting synchronization cost; e.g., by changing contention. Further, DeNovo may inherently see lower traffic because it has word-based coherence state and does not transfer locations in a cache line known to be invalid. Since these overheads are orthogonal to the synchronization focus of this chapter, we do not discuss them here, but they do affect our results in Section 4.5.

4.3.1 Locks

Atomicity synchronization ensures that a sequence of instructions executes as a single, indivisible unit [110], and its most common form is a lock. Here we explore the commonly used single variable based Test-and-Test-and-Set (TATAS) lock as well as the distributed array (or list) lock.

TATAS Locks

For individual TATAS lock acquire and release, we identify their linearization points at a successful Test-and-Set instruction for acquire, and the release write of a lock for release.

Linearization cost: When there are no or only two competing cores, the linearization cost for MESI and DeNovo is simply a two-hop or three-hop write miss acquiring the Modified (Registered) state from the shared LLC or the other core's cache respectively. As the number of contending cores increases, MESI is expected to have increasingly high linearization cost, since invalidating all cached copies increases the release write latency for the lock. While the release write may be overlapped with subsequent instructions at the releasing core, its latency is on the critical path of the next acquirer; the release write will be made visible to the next acquirer only after invalidations are complete for all cores (to ensure the write atomicity requirement for sequential consistency). In terms of network traffic, MESI incurs increasingly high network traffic for invalidation and acknowledgment messages. On the other hand, DeNovoSync/0 has no invalidation message overheads and only needs point-to-point registration transfer. Thus, DeNovo's linearization cost is expected to be independent of the number of competing cores and generally lower than MESI's.

Pre-linearization cost: Pre-linearization cost occurs only for acquires and consists of reads that perform preliminary checks to determine if it is worth proceeding to the linearization point. For MESI, waiting cores efficiently spin on a cached copy, incurring read misses only after the lock is released. For DeNovo, every synchronization read incurs a miss, unless the lock is Registered. Since registration can be revoked by remote synchronization reads, read misses by such false R-R races can be a source of performance inefficiencies for DeNovo. However, unless critical sections are very long, the percentage of false races relative to the total registration transfers will not be high since writes (CAS acquire and write release) by a winning core will frequently intervene between pre-linearization reads (which will trigger legitimate registration transfer). If critical sections are long and there are multiple waiting cores, DeNovoSync0 will suffer excessive ping-ponging of registrations between these cores, but DeNovoSync is expected to reduce this inefficiency through its hardware backoff.

Overall, for most cases, we expect that the linearization cost will dominate since it is on the critical path of the lock handoff; therefore, DeNovoSync/0 will perform better than MESI for execution time and network traffic. An exception is the case of long critical sections with multiple waiters, where DeNovoSync0's higher pre-linearization costs may or may not be offset by its lower linearization cost.

Array Locks

To reduce synchronization overheads for highly contended locks, array locks distribute synchronization points in space so that cores wait on different locations (array entries) in order. The linearization points are (1) a synchronization read in the acquire loop indicating the lock for the given entry is available, and (2) the release write of the next entry's lock.

Linearization cost: Array locks allocate a unique memory location for each acquiring core. Linearization involves setting of this location by the releaser and reading of the new value by the acquirer. For DeNovoSync/0, this is a simple 2 or 3 hop registration transfer for both the release and the acquire. For MESI, the critical path is similar (there are no excessive invalidations as for TATAS), but there are two subtle effects. First, the transactions are slightly more complex; e.g., the acquire involves a potential writeback to the directory and accompanying unblock messages for a blocking directory. Second, the successful acquire read is immediately followed by a write to reset the same lock so it can be reused in the next invocation. For DeNovoSync/0, this write is a hit since the lock is already registered by the acquire, but MESI needs to separately request ownership. This write latency for MESI can be on the critical path if the critical section is too small and the ensuing data accesses cannot overlap the latency. In that case, the write latency will be visible as an additional linearization cost for MESI. A read for ownership transaction in MESI would eliminate that cost, but our system does not implement such a transaction.

Pre-linearization cost: Pre-linearization includes spinning on the unique lock entry for each waiter. Since there is only one waiter per entry, both DeNovo and MESI spin on a cached copy, with minimal overhead.

Overall, for array locks, we expect comparable performance except for very small critical sections (where MESI may be worse) and higher traffic for MESI due to its more complex transactions and additional ownership request.

4.3.2 Non-Blocking Algorithms

It is common that non-blocking algorithms (1) linearize at their final CAS instruction, and (2) perform relatively many reads for equality checks to guarantee fast forward progress until the linearization point (e.g., Figure 4.2).

Linearization cost: Non-blocking algorithms provide atomicity of concurrent operations by linearizing cores that successfully execute the final CAS instruction. Similar to TATAS locks, MESI adds significant invalidation overheads for linearization cost with many competing cores, while DeNovo requires only a single point-to-point communication.

Pre-linearization cost: Non-blocking algorithms tend to have many repeated reads for equality checks on multiple synchronization variables before the linearization point. In this case where synchronization read-to-write ratio is high, DeNovo with pessimistic reader-initiated self-invalidation is likely to suffer from spurious read misses from R-R and W-R registration transfer. While MESI can perform such reads without any cost unless there is a true race, we expect DeNovo to have high pre-linearization cost proportional to the number of synchronization read accesses.

Overall, under relatively low contention, DeNovoSync/0 may offset its high pre-linearization cost with its lower linearization cost compared to MESI. Under high contention, DeNovoSync0 will likely see the negative impact of pre-linearization cost while DeNovoSync can mitigate the impact with hardware backoff.

4.3.3 Barriers

For barriers, separate linearization and pre-linearization points can be defined for arrival and departure phases. Signaling arrival (e.g., incrementing the arrived core counter for centralized barriers or reversing and propagating parent's flag for tree barriers) and departure (e.g., reversing the sense(s) and propagating the updated sense to waiting cores) belongs to linearization, while pre-linearization includes unsuccessful checks (synchronization reads) of sense/flags (in a loop).

Linearization cost: For a centralized barrier, for arrival, both MESI and DeNovo incur high linearization cost by serializing cores incrementing the arrived core counter. MESI with blocking directory may suffer from extra queuing and messaging delays than DeNovo when the counter is severely contended. On departure, the release of a centralized barrier allows many readers to proceed. For MESI, this adds invalidation related overheads to linearization cost similar to TATAS locks. For DeNovo, we expect a potentially higher cost because all the waiting readers incur serialized registrations, adding to the critical path.

The linearization cost for both MESI and DeNovo is lowered with tree barriers since these reduce the number of threads synchronizing on a given location. Trees in tree barriers can be configured with different width and depth, depending on the number of child threads synchronized for the same parent thread. Regardless of the tree structure, there is only one reader and one writer for each node in the trees: an owner thread for a given tree node and its parent thread. For arrival, the owner thread updates its node (flag) and its parent thread reads it to see if the thread has arrived and vice versa for departure. Thus, the case of a distributed tree barrier is similar to that of an array lock, and we expect similar latency for both DeNovo and MESI.

Pre-linearization cost: Pre-linearization cost for barriers comes from synchronization reads by cores waiting for their flag to be reversed. Again, MESI efficiently spins from the local cache. DeNovoSync/0 pays similar costs for distributed tree barriers with only one reader per node. For scenarios with a large number of readers (i.e., centralized barrier), DeNovoSync0 sees a negative effect on network traffic through ping-ponging registrations, while DeNovoSync partly mitigates this effect through the hardware backoff.

Overall, MESI and DeNovo perform comparably for distributed tree barriers, behaving similar to array-based locks. DeNovo, however, can potentially suffer from higher pre-linearization and linearization costs with worse performance and traffic for centralized barriers. We note, however, that centralized barriers with many readers and one writer contending for a single variable is not a scalable synchronization pattern regardless of the coherence protocol. For best absolute performance, tree barriers with limited read sharing per synchronization variable are preferred. DeNovoSync is expected to be comparable to MESI in both execution time and traffic for such scalable barriers.

4.4 Methodology

For our evaluations, We use the same set of simulator tools as used in DeNovo (Chapter 2) and DeNovoND (Chapter 3). Table 4.2 shows the key parameters of our simulated systems.

4.4.1 Simulated Systems

We evaluated the following systems:

- **MESI:** We used the GEMS implementation of the MESI protocol [88], modified to support non-blocking writes for a fair comparison with DeNovo (where writes are non-blocking by default).
- **DeNovoSync0:** DeNovoSync0 implements the protocol as described in Section 4.2.1.
- **DeNovoSync:** DeNovoSync enhances DeNovoSync0 with the hardware backoff mechanism from Section 4.2.2, using the following parameters: 9-bit backoff counter with 1-cycle default increment for 16 cores and 12-bit backoff counter with 64-cycle default increment for 64 cores.

4.4.2 Workloads

We study several synchronization kernels as well as applications from standard benchmark suites. The kernels allow analyzing the new protocols in detail and showing that they can efficiently support a large variety of synchronization patterns. The applications are dominated by data accesses, but show that DeNovo can fully support common workloads.

Synchronization Kernels

We evaluated 24 synchronization kernels covering several lock-based concurrent data structures, non-blocking data structures, and barriers.

For lock-based structures, we adapted 5 kernels from [92]: single-lock queue, double-lock queue, stack, heap, and counter. The original kernels in [92] used Test-and-Test-and-Set (TATAS) locks. We also evaluated them using the more efficient array/list based queuing locks [12]. The critical sections of most of the above kernels have a small number of data accesses for one or two shared variables (e.g., pointers to queue head/tail or stack top) which allows us to isolate protocol behaviors for synchronization accesses. To cover a larger space, we also wrote a small kernel, *large CS*, with larger critical sections of fixed length, again both with TATAS and array locks. We did not use software backoff for any of the above kernels because our preliminary experiments with TATAS based kernels showed increased benefit of DeNovo over MESI for software backoff (further discussed in Section 4.5.1).

For non-blocking data structures, we adapted 6 kernels from [92]: Michael-Scott queue, PLJ queue, Treiber stack, Herlihy stack, Herlihy heap, and FAI counter. Each kernel has a software exponential delay in the range of [128, 2048) cycles to backoff after a failed attempt.

For barriers, we evaluated a static binary tree barrier, a static tree barrier with non-binary fan-in of 4 and fan-out of 2, and a centralized sense-reversing barrier derived from pseudo codes in [110]. To examine how the variance in the amount of parallel work between barriers can affect barrier performance, we evaluated a load-balanced and an unbalanced version for each barrier.

As mentioned, we perform region-based static self-invalidation for data consistency. We manually identified memory locations protected by each synchronization function call, assigned regions for them, and inserted self-invalidation instructions before read. Kernels have only a few shared data variables (e.g., a single queue or stack entry) so we put them in one region. For applications, we manually analyzed update patterns of shared data to minimize unnecessary invalidations as we did for benchmarks in Chapter 2 and 3.

We ran 100 iterations of each kernel (1,000 for *FAI counter* due to its very small size) with dummy computations in between to spread out iterations. In each iteration, a queue, stack, or a heap kernel executes one insertion and one retrieval (or deletion) function call for one entry to/from the shared

SPLASH-2	Input	**PARSEC**	Input
FFT	m16	fluidanimate	sim_small
LU	n256	blackscholes	sim_medium
barnes	8192	swaptions	sim_small
radix	524288	canneal	sim_small
water	512	ferret	sim_small
ocean	258	x264	sim_medium
		bodytrack	sim_medium

Table 4.1: Benchmark inputs.

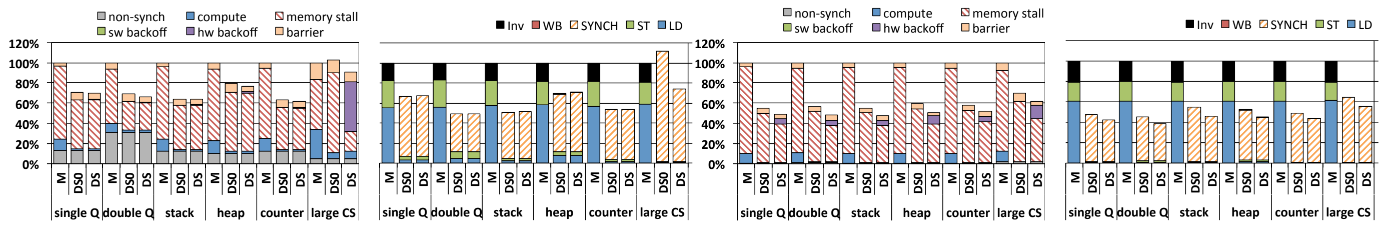
# of cores	16 cores	64 cores
Processor Parameters		
Core frequency	2 Ghz	
Memory Hierarchy and Network Parameters		
L1 data cache	32KB, 64 bytes line	
L2 (NUCA)	4MB, 16 banks	8 MB, 64 banks
	64 bytes line	
Memory	4GB, 4 on-chip controllers	
L1 hit latency	1 cycle	
L2 hit latency	28 to 68 cycles	28 to 140 cycles
Remote L1 hit latency	37 to 97 cycles	37 to 205 cycles
Memory hit latency	197 to 277 cycles	197 to 421 cycles
Network parameters	2D mesh, 16 bit flits	

Table 4.2: Simulated system parameters for DeNovoSync.

data structure, a counter kernel performs a single increment, and a barrier kernel executes two barrier instances around dummy computation. The length of the dummy computations is randomly chosen in the range of [1400, 1800) cycles for 16 cores and [6200, 6600) cycles for 64 cores, except that the unbalanced versions of barriers use [400, 2800) cycles for 16 cores and [1600, 11,200) cycles for 64 cores.

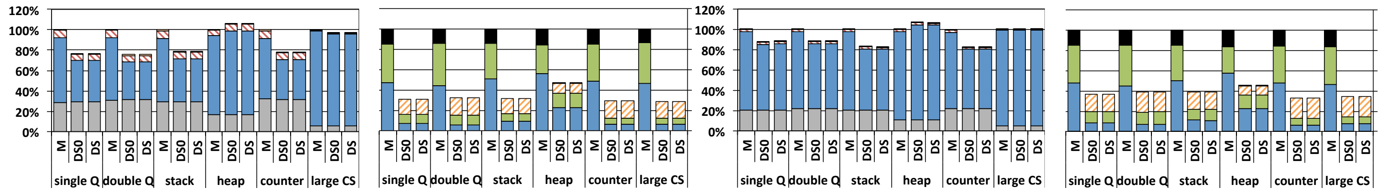
Benchmarks

We evaluated 13 benchmarks from SPLASH-2 [123] and PARSEC 3.1 benchmark suites [19]. Table 4.1 shows the benchmarks and their input parameters. The benchmarks are chosen to represent programs with different synchronization patterns including locks, barriers, producer-consumer synchronization (pipeline parallelism), and aggressive lock-free synchronization. We show results with 64 cores for all except two benchmarks. Results for *ferret* and *x264* are for 16 cores because the simulation inputs provided do not fully utilize 64 cores concurrently. In all cases, we inserted region-based static self-invalidation instructions, both in the application code and in the POSIX thread library synchronization routines that were used.



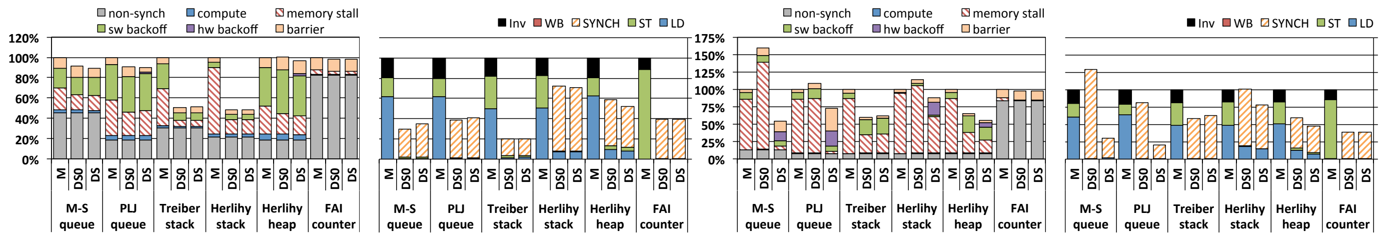
(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 4.4: Test-and-Test-and-Set (TATAS) locks based synchronization.



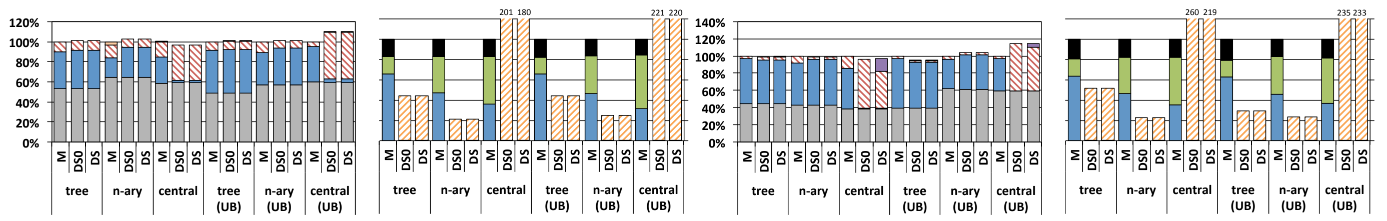
(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 4.5: Array locks based synchronization.



(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 4.6: Non-blocking algorithms.



(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 4.7: Barrier synchronization (UB = unbalanced computations).

4.5 Evaluation Results

We next validate our qualitative analysis by evaluating a variety of synchronization kernels (Section 4.5.1) and applications (Section 4.5.2).

4.5.1 Results for Synchronization Kernels

Figure 4.4, 4.5, 4.6, and 4.7 show results for synchronization kernels using TATAS locks, array locks, non-blocking algorithms, and barriers respectively. In each figure, parts (a) and (b) respectively show the execution time and network traffic for MESI (denoted M), DeNovoSync0 (denoted DS0), and DeNovoSync (denoted DS) protocols, all normalized to MESI, for a 16 core system. Parts (c) and (d) show the analogous data for a 64 core system. We measure time in cycles and network traffic in terms of flit crossings across all network links (i.e., a flit going over one network link constitutes one unit of network traffic).

Parts (a) and (c) in each figure divide the execution time into multiple components. The gray (non-synch) component is the computation time spent between iterations of the studied synchronization kernels. These non-synchronization cycles are not affected by our techniques, but changing their length may increase/decrease the contention level. The rest of the components all represent the time spent strictly within invocations of the synchronization kernels for computation and memory accesses to data and synchronization variables (as described in Section 4.4, most kernels are dominated by synchronization accesses). These components consist of compute time (1 cycle per instruction, not including software backoff cycles), memory stall time for both synchronization and data accesses inside the kernel, software backoff time, hardware backoff time (only for DeNovoSync), and barrier stall time. Barrier stall time measures the time spent in the barrier at the end of the kernels for non-barrier kernels, which can indicate possible load imbalance caused by contention in synchronization methods. Note that a large part of compute time is from spinning synchronization read accesses (cache hits), so it can vary across protocols.

Parts (b) and (d) in each figure divide the network traffic based on the message type. For MESI, we show load, store, writeback, and invalidation (including ack) messages. For DeNovo, we show data load, data store, synchronization (load, store, and RMW), and writeback messages.[3]

[3]For MESI, our system does not explicitly identify loads and stores as synchronization or data. We use fences to enforce ordering at the acquire and release synchronization points. We therefore do not separate LD/ST traffic into data and synchronization for MESI. For DeNovo, on the other hand, such information was readily available and we report it here.

Test-and-Test-and-Set (TATAS) Locks

Figure 4.4 shows that across all synchronization kernels using TATAS locks, DeNovoSync outperforms MESI in terms of execution time (31% on average) and network traffic (42% on average) for 16 and 64 cores.

MESI vs. DeNovoSync0: DeNovoSync0 outperforms MESI on both systems except for *large CS* on 16 cores. The benefit in execution time ranges from 20% to 36% (average 25%) on 16 cores and an even higher 40% to 45% (average 41%) on 64 cores. Based on the analysis in Section 4.3, this benefit occurs because MESI incurs higher linearization cost in the form of invalidation/ack latencies on the critical path from lock release to the next successful acquire. For these kernels, which have small critical sections, the linearization cost dominates pre-linearization cost as explained in Section 4.3. Further, this cost increases in larger systems when more cores need to be invalidated, explaining the higher benefit for 64 core systems. For the *large CS* kernel, on the other hand, higher pre-linearization cost from false registration transfers makes DeNovoSync0 perform slightly (3%) worse than MESI on 16 cores. On 64 cores, however, this effect is mitigated by MESI's higher linearization cost and DeNovoSync0 shows 29% reduction in execution time.

DeNovoSync0 reduces network traffic by 35% on average over MESI on 16 and 64 cores, except for *large CS* where it suffers from high read miss rates as discussed above. The main reasons for reduced traffic on DeNovo are: (1) lack of invalidation/ack traffic (which contributes about 20% of network traffic for MESI), and (2) per-word coherence granularity for DeNovo which allows sending only valid data. The latter significantly reduces response traffic for synchronization requests on DeNovo since most software pads lock variables to avoid false sharing.

DeNovoSync0 vs. DeNovoSync: DeNovoSync is comparable or better than DeNovoSync0 for all TATAS-based kernels on 16 and 64 cores (average 5% and 13% lower execution time and 14% and 16% lower network traffic). The benefit comes from delaying unnecessary synchronization read registrations, improving both network traffic and execution time through reduced contention.

Impact of lock padding: We ran all TATAS kernels without lock padding and found that most showed worse performance for MESI than the original versions due to false sharing. However, performance gaps between MESI and DeNovoSync/0 are also reduced by removing the padding because

DeNovo has to issue more read/write requests separately for locks and data in the same cache line.

Impact of software backoff: We conducted a sensitivity study to see the impact of software backoff on TATAS-based kernels. We ran the kernels with exponential software backoff in the range of [128,2048) cycles. We found that the performance gap between DeNovoSync0 and MESI increased significantly (up to 70% on 64 cores), relative to kernels without software backoff. Similar to hardware backoff, software backoff also separates failed synchronization read accesses. As a result, it reduces read misses from false races for DeNovoSync/0. On the other hand, its does not affect invalidation latency for MESI, which is the largest source of memory stall time in these kernels.

Array Locks

Figure 4.5 shows that for array lock based kernels, DeNovoSync0 and DeNovoSync show similar performance and traffic. As analyzed in Section 4.3, the single-reader design of array locks does not generate spurious registrations and does not benefit from DeNovoSync's backoff.

Compared to MESI, DeNovoSync/0 provides comparable or (up to 24%) better performance except for *heap* and reduces network traffic by 64% on average. As discussed in Section 4.3, for the most part, synchronization related pre-linearization and linearization costs for MESI and DeNovo are largely similar except for subtle effects; e.g., DeNovo saves one write miss whose latency may or may not be overlapped in MESI, depending on the critical section length. Based on detailed analysis of our experiments, we find that the overall performance of array lock kernels is therefore sensitive to the nature of the critical section computation.

Four of the six kernels (all except *large CS* and *heap*) have very small critical sections; therefore, the additional write miss for MESI adversely impacts performance, giving DeNovo an advantage. *Large CS* has a large critical section, which removes this advantage, and both MESI and DeNovo see comparable performance. *Heap* also has a high number of data accesses, but their unpredictability makes DeNovo suffer from conservative region-based static self-invalidations as follows. *Heap* re-balances its tree as entries are inserted/deleted, requiring a data-dependent traversal of the tree nodes. Without a priori knowledge of which nodes have been previously updated, DeNovo conservatively self-invalidates all nodes, resulting in unnecessary data misses as well as increasing synchronization wait times for subsequent acquirers. DeNovo therefore performs 6% and 7% worse than MESI on 16 and 64 cores respectively. This can be remedied using dynamic hardware signatures to more precisely determine what data to invalidate as in [119], but is orthogonal to the synchronization focus of this work.

Finally, DeNovo's traffic savings are for the same reasons as for the TATAS locks and saved write misses.

Non-Blocking Algorithms

Figure 4.6 shows that the non-blocking data structures exhibit varying performance depending on their access patterns and system contention. On 16 cores, both DeNovoSync0 and DeNovoSync perform comparably or better than MESI (average of 14% better execution time and 60% better traffic). On 64 cores, DeNovoSync outperforms MESI (by an average of 28% for execution time and 54% for traffic). DeNovoSync0, however, often does worse than MESI on 64 cores; e.g., up to 60% worse execution time.

MESI vs. DeNovoSync0: As discussed in Section 4.3, DeNovoSync0 suffers from high pre-linearization cost caused by unnecessary registration transfers for many synchronization reads on multiple variables. On 16 cores, this cost is offset by MESI's higher linearization cost (due to invalidations). Thus, all kernels show comparable or better performance and network traffic for DeNovoSync0. With increased number of cores, threads experience more failed equality checks before reaching the linearization point, and DeNovoSync0 is likely to suffer from increased unnecessary read registrations and misses due to interference from remote synchronization reads. For 3 of the 6 non-blocking kernels, DeNovoSync0 therefore performs worse than MESI by 28% on average on 64 cores.

DeNovoSync0 vs. DeNovoSync: Under low contention and with software backoff delay, DeNovoSync's hardware backoff does not provide visible performance boost over DeNovoSync0 (2% better on average for 16 cores). On the other hand, as contention increases on 64 cores, DeNovoSync performs much better than DeNovoSync0 (by 30% and 41% on average for execution time and network traffic respectively). Figure 4.6 shows that the improved performance directly comes from replacing memory stall time with smaller hardware backoff cycles.

Software Modifications: We observed that many non-blocking algorithms are designed to perform well on systems with a writer-initiated invalidation protocol. Since a core can spin on a cached copy on such systems, most non-blocking algorithms have repeated equality checks comparing a shared variable's current value and a core's local snapshot. These checks help non-blocking algorithms make faster progress on a failed attempt. The same checks do more harm than good for reader-initiated invalidation protocols like DeNovo. If synchronization reads occur much more frequently than writes in a synchronization loop, read registration requests will end up ping-ponging registration between readers, forcing them to miss even if shared variables are not updated at all. Since many equality checks exist only for performance but not for correctness, removing/adding some checks is safe. In our experiments, we modified *Herlihy stack* and *Herlihy heap* from [92] to reduce the number of such checks (these kernels had the most equality checks among those we tested).

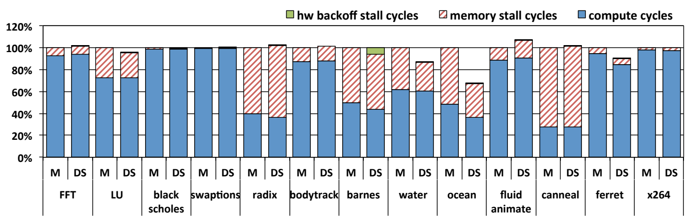
We saw that the modifications significantly shortened execution time for both MESI and DeNovo, but DeNovoSync saw much higher improvement than MESI, with 41% and 79% lower execution time on average on 16 and 64 cores compared to the unmodified version. Network traffic also reduced by up to 78% on 64 cores. Studying how implementation details of synchronization algorithms may affect their behaviors on different coherence protocols is an interesting future research direction.

Barriers

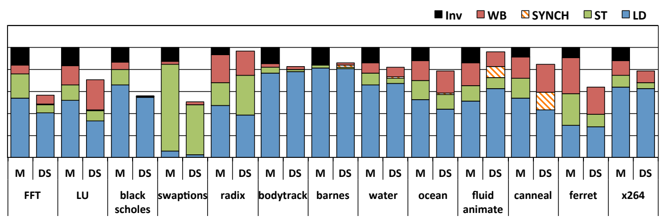
Figure 4.7 shows that DeNovo performs comparably or better than MESI for the barrier kernels studied on 16 and 64 cores except for centralized barrier with highly unbalanced computations (denoted UB) on 64 cores. For network traffic, the DeNovo protocols are better (67% on average) than MESI for the tree barriers, but much worse for the centralized barrier.

As discussed in Section 4.3, since the tree barriers exhibit a single producer and single consumer scenario for the linearization synchronization variable, all protocols behave similarly for execution time. DeNovoSync/0 sees much lower traffic for reasons similar to other kernels above.

For centralized barriers, MESI has higher linearization cost during arrival than DeNovo while DeNovo has higher linearization cost during departure, as discussed in Section 4.3. With load balanced computations, these effects offset each other and the execution times for all protocols are comparable. In case of higher load imbalance, however, DeNovo suffers from even more read registrations (and higher pre-linearization cost) than the load-balanced case, delaying the linearization point. As a result, DeNovoSync/0 performs worse than MESI by 9% and 14% on 16 and 64 cores respectively. The excessive registrations in the pre-linearization and linearization phases for DeNovo result in high traffic in both load-balanced and imbalanced cases.



(a) Execution Time.



(b) Network Traffic.

Figure 4.8: Execution time and network traffic for MESI and DeNovoSync for applications.

Nevertheless, comparing absolute performance of centralized and tree barriers, we confirmed that tree barriers are better for both our 16 and 64 core systems. Our results thus show that as long as designed in a distributed and scalable fashion, DeNovo can provide reasonable performance even for barrier synchronization (and more generally, conditional synchronization) where the synchronization accesses are mostly reads.

4.5.2 Results for Applications

Figure 4.8 shows execution time and network traffic for 13 benchmarks (*ferret* and *x264* on 16 cores, others on 64 cores) on MESI (denoted M) and DeNovoSync (denoted DS), normalized to MESI. The components for each bar are as for the synchronization kernels except that there is no separation of non-synchronization time. We see that DeNovoSync provides comparable execution time to MESI (4% better on average) – it is noticeably better for *LU, water, ocean,* and *ferret* (up to 36% better), but 7% worse for *fluidanimate*. For network traffic, DeNovoSync is 24% better than MESI on average for all benchmarks. We next discuss the applications in more detail, classified by the synchronization patterns they include.

Barrier-only: FFT, LU, blackscholes, swaptions, and *radix* use only barrier synchronization (with tree barriers). For execution time, as expected, DeNovoSync is comparable to MESI. It provides a slight advantage for *LU* since *LU* exhibits data false sharing with MESI (which DeNovo avoids due to word-level coherence [38]). DeNovoSync significantly reduces network traffic because it does not incur invalidations, load responses do not contain invalid parts of the cache line, and stores involve only registration [38].

Barriers and locks: Bodytrack, barnes, water, ocean, and *fluidanimate* use both barriers and locks. Except for *fluidanimate*, DeNovoSync shows comparable or (up to 30%) better execution time than

MESI for these applications. For *fluidanimate*, DeNovoSync is 7% worse because of the use of static self-invalidations for shared data protected by critical sections. Self-invalidating all data in writeable regions at every lock acquire results in unnecessary invalidation of valid data for this application. As discussed in the previous section for array-based lock heap in Section 4.5.1, we can reduce unnecessary read misses with more dynamic solutions. DeNovoSync shows substantial savings in network traffic for these applications as expected.

Non-blocking synchronization: canneal is a unique benchmark that synchronizes shared pointers in an aggressive lock-free loop with CAS instructions. Unlike other applications, synchronization forms a large fraction of the memory accesses in *canneal*. Again, DeNovoSync shows comparable execution time and reduced network traffic relative to MESI.

Pipeline parallelism: ferret and *x264* use pipeline parallelism, a parallel pattern that has not been previously evaluated for DeNovo. Again, relative to MESI, DeNovoSync shows comparable or better (by 5% for *ferret*) execution time, with an average network traffic reduction of 29%.

Overall, DeNovoSync efficiently supports a variety of applications, without restrictions on synchronization patterns.

4.6 Summary

The DeNovo system has shown that hardware coherence for data accesses can be made much simpler and more efficient by leveraging software information and properties like data-race-freedom. However, adding support for races on a protocol driven by race-free software could potentially undermine all the benefits of the existing systems. DeNovoSync presented in this chapter showed that leveraging access patterns of synchronization operations can help design a simple yet flexible mechanism that can support the coherence and consistency requirements of synchronization accesses. While the simple "single-reader" approach proposed for DeNovoSync0 may suffer from conservative invalidations and traffic under high contention, delaying these reads with a contention-dependent backoff as in DeNovoSync can improve performance significantly. We compared our new protocols with the state-of-the-art MESI on several synchronization constructs including those for challenging non-blocking data structures. We found our technique to be quite competitive and in many cases much better.

Chapter 5

Related Work

A vast body of work has focused on improving the shared-memory hierarchy, including coherence protocol optimizations (e.g., [79, 85, 87, 103, 116, 90]), relaxed consistency models [44, 46], using coarse-grained (multiple *contiguous* cache lines, also referred to as regions) cache state tracking (e.g., [28, 97, 127]), smart spatial and temporal prefetching (e.g., [114, 122]), bulk transfers (e.g., [14, 32, 53, 55]), producer-initiated communication [1, 75]), cache and directory design with flexible granularity [129, 77, 130], recent work specifically addressing multicore hierarchies (e.g., [16, 52, 128]), and many more. Our work is inspired by much of this literature, but our focus is on a holistic rethinking of the cache hierarchy driven by disciplined software programming models to benefit hardware complexity, performance, and power. In this chapter, we elaborate on work that is most closely related.

5.1 Software-Hardware Co-designed Systems

Recently, there has been active research in leveraging software knowledge to address inefficiencies in hardware design.

The recent SARC coherence protocol [67] also exploits the data-race-free programming model, but its goal is to improve the conventional directory-based protocol [7]. SARC self-invalidates "tear-off, read-only" (TRO) copies of data to save power. However, SARC does not eliminate directory storage overhead or reduce protocol complexity like the DeNovo system does. Also, the concept of touched bit, which plays an important role in the DeNovo system, is not present in SARC.

VIPS-M [107] improves on SARC by adopting self-invalidations, similarly to DeNovo. It uses private/shared information about data to perform self-invalidation and eliminates the directory by performing write-through at synchronization points. It implements a write-through protocol for synchronization accesses and claims that delaying the completion of a write-through helps in reducing spinning by other cores. The approach is similar to the delaying of QOLB attempts in [104]. However, VIPS-M is not clear about how it deals efficiently with synchronization algorithms with multiple synchronization variables and frequent reads to them, such as non-blocking algorithms. If many variables cause delayed write-through, they may interfere with each other's progress, causing degraded synchronization latency.[1] The authors recently proposed a synchronization mechanism [108] to address the issues with synchronization support in VIPS-M.

Callback [108] implements a directory dedicated to spin-waiting reads so that readers can be notified when a write occurs. The callback mechanism in [108] shares the same goal as DeNovoSync to provide coherence support for synchronization accesses without re-introducing writer-initiated invalidations, but it is different in the following aspects: (1) in Callback, cores bypass local caches and register callback at the LLC on every synchronization read to ensure that the read does not see stale data, even if there are no intervening writes. DeNovo allows synchronization reads to hit in local caches as long as the data is in Registered state, which guarantees up-to-date data and no intervening synchronization accesses. (2) In addition to the data vs. synchronization distinction, Callback requires a more fine-grained distinction for synchronization writes (i.e., waking up one reader or all readers) depending on the synchronization mechanism in which they are used. Such a distinction is not trivially given and requires automatic or manual semantic analysis to identify which type of synchronization writes should be used, followed by code modifications. In contrast, data-race-free memory models adopted by modern languages (the only software assumption made by DeNovoSync) imply a data vs. synchronization distinction and do not require any additional code annotation or modification. Interestingly, the evaluation in [108] shows that exponential backoff is as efficient as the callback mechanism in most cases as long as the right parameters are used. DeNovoSync addresses the flexibility issue with exponential backoff with an adaptive mechanism that adjusts backoff parameters in response to different contention levels.

Rigel [71], with a task-centric memory model [70] for accelerator architectures, exploits the popular sharing patterns in accelerator workloads to enforce coherence with explicit software-managed instructions at barriers. DeNovoD in Chapter 2 shares with Rigel the key motivation for focusing on applications with barriers only and providing software-driven coherence for such applications. However, Rigel requires that all dirty lines to the global shared cache be flushed at the end of each phase, whereas

[1]We corresponded with the authors to confirm that VIPS-M is not designed for lock-free data structures.

DeNovo can keep up-to-date data in local caches with write registration and selective self-invalidations. In addition, the overall DeNovo system, including DeNovoND and DeNovoSync in Chapters 3 and 4, supports a larger scope of applications with different sharing patterns (other than Bulk-Synchronous Parallelism) and unstructured synchronizations while keeping the coherence overheads minimal.

Cohesion [72], based on the Rigel architecture, proposes a hybrid memory model that switches between hardware and software coherence depending on the sharing patterns in accelerator applications. Cohesion uses the notion of "region" of memory to track which coherence domain (HW or SW) a region belongs to and to control coherence domain transitions. Regions in Cohesion require much more hardware support than DeNovo, as Cohesion requires coarse-grained and fine-grained region tables with mappings for *all memory locations* to handle transitions between coherence domains. DeNovo, a unified (not a hybrid) coherence solution, does not need such overheads because it uses regions only for selective self-invalidations, and its storage overheads for regions are limited to private caches. Cohesion does not address existing limitations of software and directory-based hardware coherence mechanisms, while DeNovo simplifies hardware coherence protocol design with software-coherence inspired ideas.

Min and Baer proposed a timestamp-based software-assisted coherence scheme without global coherence communication [93]. It relies on sophisticated compile-time analysis of memory dependencies, including perfect branch prediction for maintaining coherence. The overall DeNovo system takes a software-hardware co-design approach that exploits compiler/software-provided information for better complexity and efficiency of hardware coherence, while the hardware in [93] cannot guarantee correct execution without software assistance. DeNovo also allows dirty copies to stay in private caches for better data reuse, while [93] requires that writes go to a shared cache if there are potential conflicts.

More recently, TSO-CC [45] proposed a self-invalidation based coherence protocol for the TSO memory model. However, while it reduces writer-initiated invalidation overheads with self-invalidation, it also introduces many hardware mechanisms, such as timestamps and epochs, and complicated logics to maintain them.

Dir$_1$SW [58] simplifies a directory protocol that adds little complexity to message-passing hardware but efficiently supports programs written within the CICO model [78]. However, it reduces transient states by handling multiple-message requests by trapping to the software trap handlers. While CICO achieves simplicity by offloading the complexity to the software and operating systems, DeNovo fundamentally eliminates transient states in the protocol (assuming data-race freedom).

5.2 Hardware Optimizations for Coherence and Synchronization

More hardware-oriented research to improve the performance/storage efficiency of coherence and synchronization hardware also shares its goal with DeNovo and inspired it in many ways.

Atomic Coherence [121] leverages properties of nanophotonics, e.g., fast communication latency with little power, to simplify coherence protocols by eliminating races and guaranteeing atomic non-split transactions. DeNovo has shown that coherence protocols can eliminate protocol races without any restriction on substrates (it has been evaluated on a non-blocking mesh network).

Both SWEL [102] and POPS [59] use the classification of private vs. shared data to take different coherence actions for higher performance and storage efficiency. Similarly to Atomic Coherence [121], SWEL relies on a shared broadcast bus for sending invalidations more efficiently than MESI directory protocols on a point-to-point network only. While SWEL projects that its bus will scale well since it is used only for coherence traffic after 16 cores, DeNovo requires no special network substrate for simpler coherence and comparable or better performance than MESI on a 64-core system.

POPS focuses on achieving lower data and metadata access latencies and larger effective cache capacity by changing coherence protocol action based on the observed sharing pattern, i.e., whether accesses are to private or shared data. POPS is similar to DeNovo in that it simplifies coherence for private data. POPS delegates coherence handling to local caches for private data to free up space and reduce requests at the last-level cache, while DeNovo allows accesses to private data to stay in local caches as long as the data is Registered. However, POPS does not address the complexity of the underlying protocol, focusing more on cache efficiency. DeNovo aims to simplify coherence with comparable or better performance and reduced network traffic. The predictor table used in [59] to optimize coherence actions for different access patterns can be adapted for a realistic prediction scheme for direct point-to-point transfer as described in Chapter 2.

Dynamic self-invalidation [79] proactively downgrades cache lines before conflicting accesses to reduce invalidation overheads as well as access latencies for conflicting accesses. It requires additional hardware logic and storage to predict blocks for self-invalidation and adds more protocol states to existing protocols. In contrast, self-invalidation in DeNovo completely eliminates invalidation traffic while significantly simplifying the protocol.

Implicit QOLB [104] explores how Queue-On-Sync-Bit [48], a hardware distributed lock mechanism, can be applied to synchronization accesses. It explores Read-For-Ownership (RFO) [65] as an optimization for reducing write misses on synchronization accesses. RFO in the paper is defined in the context of prefetching ownership on a read before actual synchronization writes. While comparing potential optimizations with different trade-offs, the authors dismissed RFO, projecting that it will generate spurious read misses. We show that, for many synchronization algorithms where the read-to-write ratio is low or intermediate under reasonable contention, RFO can be a simple yet effective solution.

5.3 GPU Coherence Protocol Designs

Heterogeneous systems with CPU and GPGPU cores have become a promising solution that can address the energy wall while providing high throughput. What is inspiring for DeNovo is that GPU programming models for such systems share many common features and characteristics with the disciplined programming models that drive the DeNovo project. They effectively enforce structured parallelism by encapsulating computations on the GPU in separate GPU kernel invocations, which gives clear boundaries between CPU and GPU computations, and between different GPU computations. In addition, data-race-freedom is guaranteed as the CPU and GPU work on their own memory systems, and shared data is explicitly moved and flushed between them. Recently, there has been strong demand for coherent memory for CPU-GPU heterogeneous systems to make them more general-purpose. Current commercial implementations simply flush dirty data and invalidate the entire cache on synchronization for coherence, which has great potential for improvement and inspired research in more sophisticated CPU-GPU coherence design [113, 54, 101]. Applying DeNovo ideas to designing a simple and efficient coherence protocol for such systems [74] is a great starting point for the commercial adoption of DeNovo.

Temporal Coherence for GPU architectures [113] proposes a timestamp-based coherence protocol that uses synchronized counters to self-invalidate cache blocks and maintain coherence without messages. Though it eliminates unnecessary traffic by predicting how long data can stay valid, it relies heavily on the precision of the hardware prediction scheme and limits its applicability to GPU kernels with regular and predictable access patterns. DeNovo, with support for both deterministic and non-deterministic accesses and arbitrary synchronizations, puts fewer restrictions on the software supported. In addition, with registration in local caches, DeNovo never invalidates up-to-date (registered) data in local caches, whereas Temporal Coherence may if prediction fails.

QuickRelease [54] improves the performance of write-combining caches on GPUs for applications with fine-grained synchronizations. Similar to write-combining for per-word writes in DeNovo, QuickRelease leverages the release consistency model to reduce write request traffic. Its FIFO store queue can be applied to DeNovo to improve its write-combining performance. With caches divided into read-only and write-only sub-caches, QuickRelease triggers invalidations for read copies when write copies are evicted or reach the head of the FIFO queue. This incurs non-negligible overheads, as its performance results suggest, while DeNovo does not incur invalidation traffic.

Heterogeneous System Coherence (HSC) [101] uses a region-coherence scheme across CPUs and GPUs to reduce the overhead of providing coherence in integrated CPU-GPU systems. (Regions in HSC are different from regions in DeNovo; a region in HSC simply groups multiple cache blocks to a coarse-grained entry in a region directory.) While HSC reduces the coherence bandwidth required for GPU memory accesses with a MOESI-like region coherence protocol, it does not address existing issues of the protocol such as complexity and invalidation overheads.

Ongoing work in the DeNovo project is exploring how the advantages of DeNovo can be exploited in a CPU-GPU system; e.g., [74].

5.4 Software-Based Coherence

The DeNovo project reassesses earlier work on software-based coherence [34, 36, 35, 33, 4] in light of recent hardware technology and programming practices. Software-based coherence inserts invalidation instructions in a program based on compiler analysis regarding which cached copies turn stale and when. Analyses comparing software-hardware coherence [33, 4] also motivated the design of DeNovo by asserting the efficiency of software-directed invalidations. The above work showed that there is no clear winner since their performance varies with different sharing behavior; hardware coherence suffers from large invalidation overheads, especially with false sharing, while it outperforms software coherence when many writes are conditional (unpredictable at compile time). Software coherence may incur less coherence traffic but can generate extra read misses from conservative invalidations. Various mechanisms [34, 36, 35] have been proposed to improve software-based coherence by adding a directory to track sharing patterns only, using versions to determine whether data is stale, or performing invalidations only when accessed. DeNovo is different from previous approaches in that it focuses on simplifying "hardware" coherence by adopting software coherence techniques. Instead of relying on compilers, DeNovo leverages recent disciplined programming models to obtain precise information on shared accesses. Since it is a hardware protocol, hardware coherence states help DeNovo to reduce conservative invalidations (e.g., Registered state to identify up-to-date copies and never invalidate them), which could have not been avoided in software-only coherence.

5.5 Distributed Shared-Memory

Research in distributed shared-memory (DSM) has proposed a variety of techniques to provide an illusion of a coherent global address space for distributed memory systems. Without hardware support, DSM provides coarse-grained coherence using virtual memory, runtime system, and compiler support [69, 18, 60]. In order to hide high memory access overheads in distributed systems, software-based coherence for DSM focuses on exploiting relaxed consistency models. DeNovo is closely related to these approaches in that it also relies heavily on software information about memory accesses and well-defined synchronization points for coherence enforcement. A key difference between DeNovo and the DSM is that the latter is designed for distributed clusters, keeping coherence and communication at a coarse-grained page granularity and storing information about modified data in software data structures in the user space. DeNovo focuses on providing a fine-grained hardware coherence solution on tightly-coupled multicores.

A key focus of DSM models involves saving invalidation traffic by postponing the propagation of modified data until an acquire point. Lazy Release Consistency (LRC) [69] maintains the consistency of all shared data at every lock transfer. Entry Consistency (EC) [18] attempts to reduce traffic by requiring programmers to bind every shared object with a lock and transferring only the bound data objects on a lock transfer. Hardware signatures in DeNovoND, as described in Chapter 3, combine update tracking mechanisms from LRC and EC. Signatures require the identification of atomic regions with atomic effects and track only such atomic data (as in EC), but the actual tracking is performed per-core, not per-lock (as in LRC). From a programming perspective, DeNovoND is more general than EC because it does not require the binding of data with a lock throughout the duration of the program. Instead, this binding is required only for a given phase and only for data that needs to be accessed in critical sections. In addition, DeNovoND does not require updated data to be directly communicated with lock transfer since ownership states are maintained through registration. Scope Consistency (ScC) [60] attempts to relax the strict and explicit bindings between data and lock in EC; instead, it uses the "consistency scope" to implicitly associate data and the acquire/release pair that protects the data. DeNovo is similar in that it also assumes a software guarantee for data-race freedom and the association of atomic regions and sections. However, it does not distinguish between non-shared and shared (atomic) data within critical sections, and has high update tracking overheads for all shared and non-shared modifications in a scope.

In addition to the above differences between DeNovo and the DSM models in handling data accesses, DeNovo can efficiently support arbitrary synchronizations with the DeNovoSync hardware, while the DSM protocols assume software implementations for locks and barriers only.

5.6 Other Related Work

Many ideas and design components in DeNovo are inspired by and adapted from previous research in related areas. For hardware signatures in DeNovo, we leverage much prior work on Bloom filters that has recently been widely used for access tracking [125, 31, 112]. Typical prior usage, however, uses filters in the range of 1K to 2K bits. DeNovoND achieves competitive performance with 256 bits with commensurately lower space and computation overheads because its key domain is limited to atomic addresses. The distributed queue-based lock mechanism for DeNovoND described in Chapter 3 is designed based on the Queue-On-Sync-Bit (QOSB) lock [48]. The DeNovoND lock implementation is different from QOSB in that it reuses LLC data banks to store queue entries, eliminating additional storage overheads for shadow lines in QOSB.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

This thesis proposes a novel shared-memory hardware that provides complexity-, performance-, and power-efficient coherence and consistency support with a software-hardware co-design approach. The benefits of the DeNovo system are as follows:

- **Complexity:** DeNovo eliminates subtle races and numerous transient states (major sources of complexity) in coherence protocols by exploiting the data-race-freedom guarantee from software. The resulting DeNovo protocol with exactly three stable states (*Invalid*, *Valid*, and *Registered*) is much easier to verify than a conventional hardware MESI protocol. Extending the baseline DeNovoD to DeNovoND and DeNovoSync to support a broader class of applications did not introduce any additional states to the protocol, incurring minimal hardware overheads.
- **Performance:** The DeNovo systems have consistently shown comparable or better performance than the state-of-the-art MESI protocol. The performance results attest that DeNovoD and DeNovoND self-invalidations for data accesses based on static regions and signatures effectively represent dynamic updates without being overly conservative. With flexible communication and direct cache-to-cache transfer, DeNovoD for deterministic codes achieves up to 79% better memory stall time than MESI. DeNovoSync provides insightful analysis on how different synchronization mechanisms will perform on DeNovoSync and MESI. DeNovoSync outperforms MESI for most of the synchronization mechanisms studied, with lower linearization (ordering) cost from the lack of invalidation overheads.

- **Energy Efficiency:** DeNovo replaces writer-initiated invalidations with local self-invalidations for potentially stale data. For more precise self-invalidations, DeNovo uses software knowledge about which memory regions will be read or written. The self-invalidation scheme enables DeNovo to eliminate sharer's lists in directories and invalidation and acknowledgment message traffic, which are major sources of energy inefficiency. Performance benefits driven by fewer misses from the lack of false sharing and the flexible communication optimization can be also translated into lower energy consumption for DeNovo. In addition, per-word coherence granularity helps reduce network traffic by allowing only valid words in a cache line to be supplied on misses. Adding support for non-deterministic data accesses and synchronization accesses on DeNovo efficiently keeps the network traffic overheads from signatures and synchronization races minimal. As a result, all the DeNovo systems show significantly reduced traffic compared to MESI (36% for DeNovoD, 33% for DeNovoND, and 50% for DeNovoSync, on average), which can be translated directly into energy savings.

- **Comprehensiveness:** The key insight in the thesis is "separating concerns" for different classes of memory accesses. The thesis showed how the DeNovo system evolved from focusing on supporting disciplined data accesses with software-provided metadata to providing an efficient hardware solution for synchronization accesses. Finally, by merging the DeNovo systems presented in the thesis, we seek a comprehensive coherence solution for software with all levels of discipline (described in detail as future work in Section 6.2.2).

Overall, this thesis showed the strong potential for a holistic co-design of shared-memory software and hardware. Exploiting desirable software properties allowed the elimination of unscalable overheads from coherence protocol design. The resulting protocol is not only simpler but also provides comparable or better performance and network traffic than the state of the art. Such complexity and performance advantages could be maintained throughout multiple extensions of the system by strategically broadening the target software classes.

6.2 Open Questions and Future Directions

Some important open questions and future directions are as follows:

6.2.1 DeNovo Compiler Framework

As a software-driven system, DeNovo relies heavily on software information being conveyed to the hardware for its design. For the DeNovo systems discussed in the thesis, we manually identify such information (e.g., region and effects) and annotate codes with low-level simulator-specific instructions to pass it to the simulator. Since we manipulated the simulator directly, a compiler interface was not required. For DeNovo to be a realistic solution, however, it needs full compiler support that defines how the software information is identified and represented in codes and then translated into hardware instructions.

First, a DeNovo compiler framework must translate programmer-provided annotations into special DeNovo ISA instructions. Programmer interfaces can be added in the form of library calls or keywords; modern languages already provide a way to distinguish between data and synchronization accesses in source codes. The DeNovo compiler then passes such information to the hardware using the DeNovo ISA, and the hardware will in turn take corresponding coherence actions. For example, DeNovo needs instructions to define regions, issue self-invalidations, and distinguish between data and non-data (synchronization) accesses. Chapter 2 describes at a high level how these instructions will be defined and implemented.

Second, the DeNovo compiler may implement analysis engines that automatically infer useful software information such as region information and data/non-data access distinction. Providing automated tools for extracting software information will significantly relieve the burden of manually annotating codes on programmers, and help DeNovo to be more readily adopted. However, automated tools pose a potential performance risk because they are more conservative and produce less optimal results than manual annotations.

For DeNovo, a region inference engine must identify which memory locations a phase or a critical section reads or writes. While DeNovo can always correctly run applications by making conservative assumptions about what to invalidate, it may not be able to offer comparable or better memory stall time and network traffic compared to MESI if valid data is conservatively included in a writeable region and unnecessarily invalidated. Research on improving the efficiency and precision of program analysis and inference has a long history [39, 98]. Recent research has shown that a combination of static analysis and runtime mechanism is more efficient in capturing dynamic memory behaviors than static analysis relying solely on source codes (or binary) [62, 106, 111].

Third, when source codes are not available to be augmented with DeNovo annotations, there needs to be a way to determine when and what to invalidate only from binaries. DeNovo must deal with the following three cases. (1) Distinguishing between data and synchronization accesses in binaries could be done by checking the types of memory instructions. Many architectures (e.g., ARM, Itanium) require atomic instructions to be used for proper synchronization ordering. In this case, self-invalidation instructions can simply be inserted after atomic instructions for the coherence of data accesses. Regarding what to invalidate, various binary analysis and runtime techniques can be used to identify DeNovo regions, but we can always provide coherence by being conservative and invalidating all non-registered copies. Therefore, DeNovo as of now, can fully support binaries for these architectures. (2) Some architectures (e.g., PowerPC, Alpha) with relaxed memory models require fence instructions to enforce memory access ordering. We can take fence instructions as synchronization markers, and perform self-invalidations and wait for pending registrations around them for data accesses. For synchronization accesses, distinguishing synchronization read requests from data read requests is more difficult than in the first case because synchronization read requests may not accompany fences. Based on the observation that shared-memory synchronization is always built with both reads and writes to a shared location, we can analyze binaries to identify synchronization write accesses first (using fences) and synchronization variables from them, and then search for read accesses to the same variables. (3) If fences or atomic instructions are not required for either synchronization reads or writes due to a stricter hardware memory model (e.g., x86), we can either conservatively invalidate on every read (treating all memory accesses as data) or request registration on every read (treating all memory accesses as synchronization). This will entail a significant performance penalty from virtually bypassing private caches in many cases, but the registration mechanism in DeNovo still allows cores to reuse cached Registered copies in a single-reader case, which is very common. One of the important open questions for DeNovo is how we can lower our base software assumptions and support legacy binaries without compromising performance.

It is an important future work for DeNovo to explore existing static/runtime techniques for program analysis and build an efficient analysis framework for region inference and data/non-data distinction for DeNovo.

6.2.2 DeNovoAll as a Complete Coherence Solution

As described in the introduction, each DeNovo system in the previous chapters focuses on a subclass of memory accesses for optimal design and performance. Merging them into one system with complete coherence support will be the final goal of the DeNovo project.

Such a system (tentatively DeNovoAll) aims to provide a comprehensive and adaptive coherence solution for *any* shared-memory programs with one assumption – the data-race-free memory model. Since it is common for software to be assembled from separately written and compiled components, DeNovoAll will allow programs to have “mixed disciplines” and adaptively provide optimal coherence solutions for different parts.

For data accesses, DeNovoAll can switch between three different coherence mechanisms depending on the level of discipline: (1) region-based invalidations, as proposed in Chapter 2 for disciplined data accesses whose access patterns can be statically identified and conveyed in the form of regions to the hardware, (2) signature-based invalidations, as proposed in Chapter 3 for data accesses with more dynamic and unpredictable access patterns, and (3) non-selective (flush-all) invalidations for undisciplined data accesses without any software- or hardware-provided information. All the above mechanisms invalidate potentially stale read-only data and never flush up-to-date registered data. For synchronization accesses, we can reuse the DeNovoSync protocol, as proposed in Chapter 4, assuming the distinction between data and synchronization accesses.

Preliminary evaluation results reinforce the effectiveness of the coherence schemes described in Chapters 2 and 3. A set of benchmarks from SPLASH-2 and PARSEC that we evaluated performed consistently better when region-based invalidations were enabled than when no software information was provided and non-selective invalidations were used. In addition, certain benchmarks (e.g., *fluidanimate*) perform even better when they can switch between region-based and signature-based invalidations in different code sections than when they can only use region-based invalidations, because they can benefit from more selective and precise invalidations for irregular accesses. Fully implementing and evaluating DeNovoAll with a variety of software including operating system kernels will be an important future work.

6.2.3 DeNovoSync as Synchronization Support for Related Systems

As described in Chapter 5, most of the closely related work without writer-initiated invalidations assumes that data and non-data (synchronization) accesses are distinguished and provides different solutions for these two classes. These systems tend to focus on eliminating writer-initiated invalidations for *data accesses* using safety properties such as data-race freedom and compile-time knowledge about memory access patterns. For synchronization accesses, many assume specialized implementation for high-level synchronization mechanisms or non-cacheable synchronization variables. This approach could require extra software effort, limit the types of synchronizations supported, or a incur performance penalty from bypassing caches. To our knowledge, no previous writer-invalidation-free system has supported arbitrary synchronization with cacheable synchronization variables. We expect that the ideas underlying DeNovoSync for synchronization can be integrated into other writer-invalidation-free systems to improve the efficiency and generality of their supported synchronization mechanisms. A promising candidate is CPU-GPGPU heterogeneous systems, where kernels on GPUs are expected to be data-race-free between GPUs. Only atomic instructions are allowed within a kernel for communication, and they are often implemented with special hardware that bypasses private caches. Adapting DeNovoSync support for synchronization accesses to such heterogeneous systems to improve atomic instructions would be an interesting future work.

6.2.4 Protocol-Aware Algorithm Design

During the performance evaluation for the DeNovo system, we observed that the efficiency of coherence protocols can be determined by the programmer's decisions on how to coordinate shared-memory accesses. More importantly, we found that many parallel algorithms are written and optimized assuming the writer-initiated invalidation coherence protocol. Such algorithms tend to have many shared reads for which it is difficult to know exactly the last write seen by them at compile time. For example, DeNovoND suffers from extra invalidations for shared data that is written once and read-only afterwards within critical sections (Section 3.7.3). DeNovoSync also incurs unnecessary read misses for repeated equality checks in non-blocking algorithms (Section 4.5.1). As discussed in previous sections, such reads are not an essential part of the algorithms, and software can be easily modified to reduce their number and improve its performance on the DeNovo system (and on MESI). It would be an interesting future work to explore in detail how common programming practices may affect system performance with different coherence protocols and how software can be designed in a more protocol-aware or protocol-neutral fashion for more performance consistency.

6.2.5 Support for Commercial Workloads

The DeNovo protocol has been evaluated extensively with scientific and engineering workloads that exploit data and task parallelism found in the algorithms, e.g., numerical/financial simulation, image processing, graph analysis, and others. While this class of applications comprises a large portion of shared-memory software in the market, there are more “commercial” workloads such as databases, Web servers, and others. Previous research [89, 15] has shown that these commercial workloads show significantly different characteristics from numerical workloads in terms of execution and cache performance. Frequent I/O and user interactions lead to more OS activity in commercial workloads. Unless the OS is written in a disciplined way (providing metadata) to leverage DeNovo features, the DeNovo system may have to resort to conservative coherence solutions for undisciplined OS codes. User interface codes in these workloads also show very dynamic, event-driven behaviors, which makes it hard to obtain precise region information. This may degrade overall performance for commercial workloads (but not correctness). In contrast, commercial workloads often use independent threads with minimal synchronization and large private working sets. Such access patterns require minimal coherence and consistency support. DeNovo can provide comparable performance and network traffic for such workloads compared to conventional hardware protocols, which tend to over-provision their design for arbitrary sharing. In summary, we project that commercial workloads have both advantageous and disadvantageous performance factors for DeNovo. Exploring and evaluating realistic commercial workloads on DeNovo is an important part of our future work.

6.2.6 Modeling Energy Efficiency

In the thesis, we indirectly measured the energy efficiency of DeNovo and MESI protocols with coherence network traffic, miss rates, and execution time. Measuring and comparing energy efficiency more precisely using a dedicated energy model is an important future work. While we project that energy modeling will produce results consistent with our current evaluation, it will enable us to more systematically understand how DeNovo affects overall energy efficiency. The attempt at energy modeling in [74] shows that it could be successfully applied to a DeNovo system.

6.2.7 Evaluating the Impact of Context Switches

In Sections 2.5 and 3.4.3, we discussed support for context switches on DeNovoD/DeNovoND and its potential performance impact. We project that the existence of context switches will not cancel out DeNovo's performance advantages over MESI for the reasons we described in these sections. However, we wish to more thoroughly investigate how context switches may potentially impact performance for DeNovo, especially in terms of cache interference. Evaluating the mechanisms described in the thesis to support context switches and mitigate their overheads will provide a more comprehensive evaluation of the DeNovo system.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, HPCA '97, 1997.

[2] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: Verifying memory coherence in the cray x1. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, 2003.

[3] M. E. Acacio, J. González, J. M. García, and J. Duato. Owner prediction for accelerating cache-to-cache transfer misses in a cc-numa architecture. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, 2002.

[4] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of hardware and software cache coherence schemes. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, 1991.

[5] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, Aug. 2010.

[6] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[7] S. V. Adve and M. D. Hill. Weak ordering&mdash;a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, 1990.

[8] V. S. Adve and L. Ceze. *Workshop on Deterministic Multiprocessing and Parallel Programming, U-Washington*, 2009.

[9] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, 1989.

[10] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, 2009.

[11] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, 2009.

[12] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1), 1990.

[13] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. Sharc: Checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[14] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. Empirical evaluation of the cray-t3d: A compiler perspective. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, 1995.

[15] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, 1998.

[16] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A new last level cache architecture with global block priority. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, 2007.

[17] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, 2009.

[18] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Compcon Digest of Papers.*, 1993.

[19] C. Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, 2011.

[20] C. Bienia and K. Li. Fidelity and scaling of the parsec benchmark inputs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, 2010.

[21] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[22] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, 1995.

[23] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, 2009.

[24] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, 2011.

[25] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[26] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14*, ESA'06, 2006.

[27] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent Collections. *Sci. Program.*, 18(3-4), 2010.

[28] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, 2005.

[29] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, 1977.

[30] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh, and J. Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, 2013.

[31] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, 2006.

[32] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, 1994.

[33] Y.-C. Chen and A. V. Veidenbaum. Comparison and analysis of software and directory coherence schemes. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, 1991.

[34] Y.-C. Chen and A. V. Veidenbaum. A software coherence scheme with the assistance of directories. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, 1991.

[35] H. Cheong and A. V. Vaidenbaum. A cache coherence scheme with fast selective invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, 1988.

[36] H. Cheong and A. Veidenbaum. A version control approach to cache coherence. In *Proceedings of the 3rd International Conference on Supercomputing*, ICS '89, 1989.

[37] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel sah k-d tree construction. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 2010.

[38] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[39] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, 1977.

[40] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault. Mpads: Memory-pooling-assisted data splitting. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, 2008.

[41] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009.

[42] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer &Amp; Processors*, ICCD '92, 1992.

[43] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, 1998.

[44] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, 1991.

[45] M. Elver and V. Nagarajan. Tso-cc: Consistency directed cache coherence for tso. In *IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[46] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, 1990.

[47] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures, 2007.

[48] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, 1989.

[49] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, 1988.

[50] N. Gustafsson. Axum: Language Overview. Microsoft Language Specification, 2009.

[51] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[52] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[53] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo. Ap1000+: Architectural support of put/get interface for parallelizing compiler. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, 1994.

[54] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood. Quickrelease: A throughput-oriented approach to release consistency on gpus. In *IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[55] J. Heinlein, J. Bosch, R.P., K. Gharachorloo, M. Rosenblum, and A. Gupta. Coherent block data transfer in the flash multiprocessor. In *Proceedings., 11th International Parallel Processing Symposium*, ISPP '97, 1997.

[56] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[57] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, PPOPP '90, 1990.

[58] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessor. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, 1992.

[59] H. Hossain, S. Dwarkadas, and M. Huang. Pops: Coherence protocol optimization for both private and shared data. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[60] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, 1996.

[61] Intel. The SCC Platform Overview, 2010.

[62] J. C. Jenista, Y. H. Eom, and B. Demsky. Ooojava: An out-of-order approach to parallel programming. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, 2010.

[63] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, 1995.

[64] A. Kägi, D. Burger, and J. R. Goodman. Efficient synchronization: Let them eat qolb. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, 1997.

[65] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, 1985.

[66] S. Kaxiras and J. R. Goodman. Improving cc-numa performance using instruction-based prediction. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, 1999.

[67] S. Kaxiras and G. Keramidas. Sarc coherence: Scaling directory cache coherence in performance and power. *IEEE Micro*, 30(5):54–65, 2010.

[68] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, 2011.

[69] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, 1992.

[70] J. Kelm, D. Johnson, S. S. Lumetta, M. Frank, and S. Patel. A task-centric memory model for scalable accelerator architectures. In *18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[71] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[72] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: A hybrid memory model for accelerators. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[73] R. Komuravelli. Verification and Performance of the DeNovo Cache Coherence Protocol. Master's thesis, University of Illinois at Urbana-Champaign, 2010.

[74] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. Adve. Stash: Have your scratchpad and cache it too. In *To appear in the 42nd International Symposium on Computer Architecture*, ISCA-42, 2015.

[75] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, 1995.

[76] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[77] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[78] J. R. Larus, S. Chandra, and D. A. Wood. Cico: A practical shared-memory programming performance model. In *Workshop on Portability and Performance for Parallel Processing*, 1993.

[79] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, 1995.

[80] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[81] F. Liu and Y. Solihin. Understanding the behavior and implications of context switch misses, 2010.

[82] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[83] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, 2002.

[84] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, 2005.

[85] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, 2003.

[86] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[87] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, 2003.

[88] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[89] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *SIGOPS Oper. Syst. Rev.*, 28(5):145–156, 1994.

[90] L. Menezo, V. Puente, and J.-A. Gregorio. Flask coherence: A morphable hybrid coherence protocol to balance energy, performance and scalability. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '15, 2015.

[91] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, 1996.

[92] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51:1–26, 1998.

[93] S. L. Min and J.-L. Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):25–44, January 1992.

[94] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization*, IISWC 2008, 2008.

[95] M. Mitzenmacher. Compressed bloom filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, 2001.

[96] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, 1991.

[97] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, 2005.

[98] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. 1999.

[99] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009.

[100] Oracle. Java Language and Virtual Machine Specifications.

[101] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[102] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. Swel: Hardware cache coherence protocols to map shared data onto shared caches. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, 2010.

[103] A. Raghavan, C. Blundell, and M. M. K. Martin. Token tenure: Patching token counting using directory-based cache coherence. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, 2008.

[104] R. Rajwar, A. Kagi, and J. Goodman. Improving the throughput of synchronization by insertion of delays. In *Sixth International Symposium on High-Performance Computer Architecture*, HPCA-6, 2000.

[105] A. D. Robison. Intel® threading building blocks (tbb). In *Encyclopedia of Parallel Computing*, pages 955–964. 2011.

[106] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, 2010.

[107] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, 2012.

[108] A. Ros and S. Kaxiras. Callback: Efficient synchronization without invalidation with a directory just for spin-waiting. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[109] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, 2007.

[110] M. Scott. *Shared Memory Synchronization.* Synthesis Lectures on Computer Architecture. 2013.

[111] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid static–dynamic analysis for statically bounded region serializability. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[112] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, 2008.

[113] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt. Cache coherence for gpu architectures. *Micro, IEEE*, 34(3):69–79, May 2014.

[114] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, 2006.

[115] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, and D. A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Trans. Parallel Distrib. Syst.*, 13(6):556–578, 2002.

[116] K. Strauss, X. Shen, and J. Torrellas. Flexible snooping: Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, 2006.

[117] S. Subramaniam, S. C. Steely, W. Hasenplaugh, A. Jaleel, C. Beckmann, T. Fossum, and J. Emer. Using in-flight chains to build a scalable cache coherence protocol. *ACM Trans. Archit. Code Optim.*, 10(4):28:1–28:24, 2013.

[118] H. Sung and S. V. Adve. Supporting Arbitrary Sychronization without Writer-Initiated Invalidations. In *To appear in Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '15, 2015.

[119] H. Sung, R. Komuravelli, and S. V. Adve. Denovond: efficient hardware support for disciplined non-determinism. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, 2013.

[120] H. Sung, R. Komuravelli, and S. V. Adve. Denovond: Efficient hardware for disciplined nondeterminism. *IEEE Micro*, 34(3):138–148, 2014.

[121] D. Vantrease, M. H. Lipasti, and N. Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[122] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, 2005.

[123] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, 1995.

[124] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Des. Test*, 7(4):13–25, 1990.

[125] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, 2007.

[126] J. Zebchuk, B. Falsafi, and A. Moshovos. Multi-grain coherence directories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[127] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, 2007.

[128] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[129] H. Zhao, A. Shriraman, and S. Dwarkadas. Space: Sharing pattern-based directory coherence for multicore scalability. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, 2010.

[130] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas. Protozoa: Adaptive granularity cache coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.